UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group The Data Science (DICE) group

PhD DISSERTATION

Submitted to the The Data Science (DICE) group Research Group
in Partial Fullfilment of the Requirements for the Degree of

PhD in Computer Science

Time-Efficient Link Discovery for Data-Driven Applications

by
Kleanthi Georgala

Thesis Supervisor:
Prof. Dr. Axel-Cyrille Ngonga Ngomo

Paderborn, September 13, 2021

Declaration

Translation from german:

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original declaration text in german:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

[illegible]

Athens, 26.02.2021

Abstract.
Over recent years, the Linked Data Web has grown to contain billions of facts distributed over thousands of Knowledge Bases (KBs) [12]. For example, the English version of the DBpedia Knowledge Base currently describes 4.58 million things, including 1.4 million persons, 735 thousand places, 411 thousand creative works, 241 thousand organizations and 251 thousand species. Datasets such as the LSQ [172] and LinkedGeoData now consist of more than 1.3 billion triples respectively. A direct consequence of the availability of this large amount of data in Resource Description Framework (RDF)[a] is the heightened requirement for efficient and effective Link Discovery (LD) algorithms, which compute links between RDF Knowledge Graphs (KGs). A plethora of approaches have been developed for this purpose over recent years. These approaches address the challenge of effectiveness by providing solutions driven by Machine Learning (ML) techniques ranging from genetic programming to probabilistic models. In addition to addressing the need for accurate links, Link Discovery frameworks need to address the challenge of time efficiency. This challenge comes about because of the sheer size of Knowledge Bases that need to be linked. Under the declarative representation paradigm, most Link Discovery frameworks rely on atomic or complex Link Specifications (LSs) to determine candidates.
In this thesis, we focus on the challenge of time efficiency and we propose a set of approaches towards fast and scalable Link Discovery. We devise two families of approaches:

1. approaches for optimizing the efficiency of atomic similarities for Link Discovery; and

2. approaches towards the fast execution of complex similarities and Link Specifications.

Regarding the first set of approaches, we are motivated by the absence of fast approaches for linking event data and the current performance of semantic string similarities in linking frameworks. The second family is built upon time-efficient Link Discovery approaches that operate under time and space constraints, and the absence of planning approaches, which exploit global knowledge about the execution of Link Specifications.

[a]`https://www.w3.org/RDF/`

Abstrakt.

In den letzten Jahren ist das Linked Data Web zu einer größe von mehreren Milliarden Fakten angewachsen, die über tausende Wissensbasen verteilt sind [12]. Die aktuelle englische Version der DBpedia Wissensbasis beschreibt beispielsweise 4.58 Millionen Dinge, darunter 1.4 Millionen Personen, 735 Tausend Orte, 411 Tausend kreative Arbeiten, 241 Tausend Organisationen und 251 Tausend Spezies. Datensätze wie LSQ [172] and LinkedGeoData bestehen mittlerweile aus mehr als 1.3 Milliarden Tripeln. Eine direkte Konsequenz der Verfügbarkeit dieser großen Menge an Daten im Resource Description Framework (RDF)[1] ist der wachsende Bedarf an effektiven und effizienten Link Discovery (LD) Algorithmen, die die Verbindungen (Links) zwischen den RDF Wissensgraphen (KGs) erstellen. Eine Unmenge an Ansätzen wurde in den letzten Jahren zu diesem Zweck entwickelt. Diese Ansätze verwenden vor allem Algorithmen des maschinellen Lernens – von Wahrscheinlichkeitsmodellen bis zu genetischer Programmierung – um die angestrebte Effektivität zu erreichen. Durch die schiere Größe der Wissensbasen wird neben der Generierung akkurater Links die Zeiteffizienz zu einer Herausforderung für LD Frameworks. Die meisten dieser Frameworks basieren auf atomaren oder komplexen Link Specifications (LSs), um Kandidaten für einen Link zu identifizieren.

Diese Arbeit behandelt die Herausforderung der Erstellung zeiteffizienter Linking Algorithmen. Wir präsentieren eine Menge von Ansätzen, die schnelles und skalierbares LD ermöglichen. Wir unterscheiden dabei zwei Untermengen:

1. Ansätze zur Optimierung der Effizienz atomarer LSs und
2. Ansätze zur schnelle Ausführung komplexer Ähnlichkeiten und LSs.

Die Entwicklung der erste Untermenge wird durch das Fehlen von schnellen Ansätzen zum Verknüpfen von Ereignisdaten und der derzeitigen Performanz von semantischen Ähnlichkeiten für Zeichenketten in LD frameworks motiviert. Die zweite Menge besteht aus zeiteffizienten LD Ansätzen, die mit Zeit- und Speicherbeschränkungen umgehen können, sowie Planungsalgorithmen, die globales Wissen über die Ausführung von LSs verwenden und bisher fehlten.

[1] `https://www.w3.org/RDF/`

Selected Publications

- **Scalable Link Discovery for Modern Data-Driven Applications** by Kleanthi Georgala in Proceedings of The 15th International Semantic Web Conference (ISWC2016) 2016, Doctoral Consortium Track, Kobe, Japan, 17. October - 21. October 2016
- **An Efficient Approach for the Generation of Allen Relations** by Kleanthi Georgala, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo in Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI) 2016, The Hague, 29. August - 02. September 2016
- **An Evaluation of Models for Runtime Approximation in Link Discovery** by Kleanthi Georgala, Michael Hoffmann, and Axel-Cyrille Ngonga Ngomo in Proceedings of the International Conference on Web Intelligence, 2017. Received the “Best Student Research Paper” award.
- **Dynamic Planning for Link Discovery** by Kleanthi Georgala, Daniel Obraczka, and Axel-Cyrille Ngonga Ngomo in The Semantic Web, ESWC 2018, Lecture Notes in Computer Science
- **Applying edge-counting semantic similarities to Link Discovery: Scalability and Accuracy** by Kleanthi Georgala, Mohamed Ahmed Sherif, Michael Röder and Axel-Cyrille Ngonga Ngomo in Proceedings of the 15th International Workshop on Ontology Matching 2020 (OM-2020), collocated with the 19th International Semantic Web Conference ISWC-2020, 1. November - 6 November 2020, Virtual Conference
- **LIGER - Link Discovery with Partial Recall** by Kleanthi Georgala, Mohamed Ahmed Sherif and Axel-Cyrille Ngonga Ngomo in Proceedings of the 15th International Workshop on Ontology Matching 2020 (OM-2020), collocated with the 19th International Semantic Web Conference ISWC-2020, 1. November - 6 November 2020, Virtual Conference
- **A Systematic Survey on String Similarity Joins for Link Discovery** by Kleanthi Georgala and Axel-Cyrille Ngonga Ngomo (under review for the Journal of Web Semantics)
- **LIMES - A Framework for Link Discovery on the Semantic Web** by Axel-Cyrille Ngonga Ngomo, Mohamed Ahmed Sherif, Kleanthi Georgala, Mofeed Hassan, Kevin Dreßler, Klaus Lyko, Daniel Obraczka, and Tommaso Soru. KI-Künstliche Intelligenz, German Journal of Artificial Intelligence - Organ des Fachbereichs "Künstliche Intelligenz" der Gesellschaft für Informatik e.V. (2021)

Acknowledgments

I would like to thank my supervisor, Dr. Axel-Cyrille Ngonga Ngomo for his continuous guidance and support through my Ph.D. years. His advice, insight and knowledge helped me shape the researcher I am today. But most importantly, I want to thank him for being kind, generous and patient. Axel, if I had to do my Ph.D. all over again, I would never imagine having someone else apart from you as my supervisor.

I would like to thank my colleagues in AKSW at the University of Leipzig and the Data Science Group at the University of Paderborn, for their support, advice and feedback. Our endless talks and discussions helped be through the toughest moments and guided me to become a better researcher.

Finally, I would like to thank my parents and my brother for their endless love and support. Thank you for being there for me. Thank you for believing in me.

Acronyms

WWW	World Wide Web
SW	Semantic Web
LOD	Linked Open Data
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
URI	Uniform Resource Identifier
IRI	Internationalized Resource Identifier
OWL	Web Ontology Language
LD	Link Discovery
OM	Ontology Matching
LS	Link Specification
ML	Machine Learning
KB	Knowledge Base
KG	Knowledge Graph
RO	Refinement Operator
SPARQL	SPARQL Protocol and RDF Query Language
W3C	World Wide Web Consortium
DAG	Directed Acyclic Graph
LSO	Least Common Subsumer
SSJ	String Similarity Join
CEP	Complex Event Processing
OM	Ontology Matching
IM	Instance Matching
LGD	LinkedGeoData
LIGER	LInk discovery with Guaranteed Expected Recall
C-RO	ReCall with Refinement Operator
RO-MA	Refinement Operator with Monotonicity Assumption
CONDOR	DynamiC Planning fOr LiNk DiscOveRy
AEGLE	Allen's intErval alGebra for Link discovEry

LCH	Leacock and CHodorow
hECATE	Edge-Counting semAntic similariTies for Link DiscovEry
hECATE-B	hECATE-Baseline
hECATE-I	hECATE with Indexing
hECATE-IF	hECATE with Indexing and Filtering
RMSE	Root-Mean-Square Error

Table of Symbols

Symbol	Explanation
(c, p, o)	RDF triple, where c is the subject of the triple p is the predicate of the triple o is the object of the triple
$\mathcal{I}$	the set of IRIs
$\mathcal{B}$	the set of all RDF blank nodes
$\mathcal{L}$	the set of all RDF literals
S	a source Knowledge Base
$\|S\|$	the size of a Knowledge Base S defined as the number of RDF triples it includes
T	a target Knowledge Base
$\|T\|$	the size of a Knowledge Base T defined as the number of RDF triples it includes
$Rel(s, t)$	a relation between two instances $s \in S$ and $t \in T$
$\mathcal{M}$	a mapping as the set $\{(s, t) \in S \times T : Rel(s, t)\}$
$\mathcal{P}$	the set of all properties
p_s	a property of an instance $s \in S$, where $p_s \in \mathcal{P}$
p_t	a property of an instance $t \in T$, where $p_t \in \mathcal{P}$
$m(s, t)$	a similarity function between s and t, w.r.t to the properties p_s, p_t: $m(s, t, p_s, p_t)$, if m is atomic: $m(p_s, p_t)$
$\sigma(s, t)$	a distance function between s and t, w.r.t to the properties p_s, p_t: $\sigma(s, t, p_s, p_t)$, if σ is atomic: $\sigma(p_s, p_t)$
θ	a similarity threshold
ϖ	a distance threshold
$\mathcal{M}^*$	a mapping as the set $\{(s, t) \in S \times T : m(s, t) \geq \theta\}$ or $\{(s, t) \in S \times T : \sigma(s, t) \leq \varpi\}$
L	a Link Specification
$\mathcal{LS}$	the set of all Link Specifications
$[[L]]$	the mapping of a Link Specification L
$L_\emptyset$	an empty Link Specification
$(m(p_s, p_t), \theta)$	an atomic Link Specification described using an atomic similarity function and a threshold

Symbol	**Explanation**
(f, ζ)	a filter, where f is either empty (ϵ) or, a similarity measure or, a combination of similarity measures, and ζ is a threshold
(f, ζ, X)	an atomic Link Specification described using a filter, with $X = S \times T$
$op(L) = \omega$	the specification operator of a complex Link Specification L
$(f, \zeta, \omega(L_1, L_2))$	a complex Link Specification described using a filter
$\varphi(L) = (f, \zeta)$	the filter of a Link Specification L
L_1	the left sub-specification or left child of a complex Link Specification L
L_2	the right sub-specification or right child of a complex Link Specification L
$\|L\|$	the size of a Link Specification L
$\|[[L]]\|$	the size of the mapping of a Link Specification L
$sel(L)$	the selectivity function that returns the selectivity of a Link Specification L
$\rho(L)$	a downward refinement operator of a Link Specification L
Λ	a Link Specification subsumed by L or a specialization of L, such that $\Lambda \in \rho(L)$, denoted $L \sqsubseteq \Lambda$
$\rho^*(L)$	the set of all Link Specifications that can be reached from a Link Specification L via ρ
k	a user-defined minimal expected recall requirement
$maxOpt$	a user-defined refinement time constraint
$\vec{S} = (\|S_1\|, \ldots, \|S_n\|)$	vector of sizes obtained by sampling S
$\vec{T} = (\|T_1\|, \ldots, \|T_n\|)$	vector of sizes obtained by sampling T
$\vec{\theta} = (\theta_1, \ldots, \theta_n)$	vector of different threshold obtained by sampling θ
$\psi(S, T, \theta)$	a fitting function, whose value at $(\|S\|, \|T\|, \theta)$ is an approximation of the runtime for the Link Specification with these parameters
$\vec{R} = (R_1, \ldots, R_n)$	a vector of measured runtimes for the parameters $\vec{S}$, $\vec{T}$ and $\vec{\theta}$
$E(\vec{S}, \vec{T}, \vec{\theta}, \vec{r})$	a local minimum of the L2-Loss
$\psi_1(S, T, \theta)$	a linear fitting function
$\psi_2(S, T, \theta)$	standard log-linear fitting function
$\psi_3(S, T, \theta)$	an interpolation of $\psi_1(S, T, \theta)$ and $\psi_2(S, T, \theta)$
V	a set of concepts
n_V or $\|V\|$	the number of concepts included in V

Symbol	**Explanation**
c_i	a concept as a set of synonyms, $\forall i \in [1, n_V]$
e_{jk}	hypernymy relation from a parent concept c_j to a child concept c_k
E	a set of directed edges $e_{jk} = (c_j, c_k)$
n_E or $\|E\|$	the number of edges included in E
$G = (V, E)$	lexical vocabulary as a directed acyclic graph
$c_j \rightarrow c_k$	c_j is a hypernym of c_k
$c_j \leftarrow c_k$	c_j is a hyponym of c_k
$root$	the unique node of $G = (V, E)$ that has no parent concept
$cs(c_1, c_2)$	a common subsumer of two concepts c_1 and c_2
$lso(c_1, c_2)$	the least common subsumer two concepts c_1 and c_2
$path(c_1, c_2)$	a directed path from c_1 to c_2 via a common subsumer $cs(c_1, c_2)$
$len(c_1, c_2)$	the length of the shortest $path(c_1, c_2)$ between two concepts c_1 and c_2
$depth_m(c_i)$	the length of the shortest path between $root$ and c_i
$depth_M(c_i)$	the length of the longest path between $root$ and c_i
D	the maximum $depth_M(c_i)$ found in $G = (V, E)$, $\forall i \in [1, n_V]$
Σ	an alphabet
r	a string as the finite sequence of characters over an alphabet Σ
Σ^*	the set of all possible strings over Σ
$\mu(r, g)$	a string similarity measure between two strings r, g
λ	a string similarity threshold
$d(r, g)$	a string similarity metric (distance) between two strings r, g
γ	a similarity metric (distance) threshold
τ	the Levenshtein distance threshold
δ	the overlap similarity threshold
$b(s)$	the beginning time of an event s
$e(s)$	the end time of an event s
$bf(s, t)$	the “before” relation between two events s and t, that is satisfied if s ends before t begins
$bfi(s, t)$	the “inverse before” relation between two events s and t, that is satisfied if t ends before s begins
$mt(s, t)$	the “meet” relation between two events s and t, that is satisfied if s ends at the same time that t begins
$mti(s, t)$	the “inverse meet” relation between two events s and t, that is satisfied if t ends at the same time that s begins

Symbol	**Explanation**
$fin(s,t)$	the "finishes" relation between two events s and t, that is satisfied if t begins before s and both events end at the same time
$fini(s,t)$	the "inverse finishes" relation between two events s and t, that is satisfied if s begins before t and both events end at the same time
$st(s,t)$	the "starts" relation between two events s and t, that is satisfied if s and t begin at the same time and s ends before t
$sti(s,t)$	the "inverse starts" relation between two events s and t, that is satisfied if s and t begin at the same time and t ends before s
$dur(s,t)$	the "during" relation between two events s and t, that is satisfied if s begins later than t and finishes earlier
$duri(s,t)$	the "inverse during" relation between two events s and t, that is satisfied if t begins later than s and finishes earlier
$eq(s,t)$	the "equal" relation between two events s and t, that is satisfied if s begins and ends at the same time as t
$ov(s,t)$	the "overlaps" relation between two events s and t, that is satisfied if s begins before t and t begins before s ends and s ends before t ends
$ovi(s,t)$	the "inverse overlaps" relation between two events s and t, that is satisfied if t begins before s and s begins before t ends and t ends before s ends
$BB^x(s,t))$	the "begin-begin" relation between two events s and t that is satisfied if $\begin{cases} s \text{ begins before } t & \text{for } x = 1 \\ s \text{ begins at the same time as } t & \text{for } x = 0 \\ s \text{ begins after } t & \text{for } x = -1 \end{cases}$
$BE^x(s,t))$	the "begin-end" relation between two events s and t that is satisfied if $\begin{cases} s \text{ ends before } t \text{ ends} & \text{for } x = 1 \\ s \text{ ends at the same time as } t \text{ ends} & \text{for } x = 0 \\ s \text{ ends after } t \text{ ends} & \text{for } x = -1 \end{cases}$
$EB^x(s,t))$	the "end-begin" relation between two events s and t that is satisfied if $\begin{cases} s \text{ ends before } t \text{ begins} & \text{for } x = 1 \\ s \text{ ends at the same time as } t \text{ begins} & \text{for } x = 0 \\ s \text{ ends after } t \text{ begins} & \text{for } x = -1 \end{cases}$

Symbol	**Explanation**
$EE^x(s,t))$	the "end-end" relation between two events s and t that is satisfied if $\begin{cases} s \text{ ends before } t \text{ begins} & \text{for } x = 1 \\ s \text{ ends at the same time as } t \text{ begins} & \text{for } x = 0 \\ s \text{ ends after } t \text{ begins} & \text{for } x = -1 \end{cases}$

Contents

1 Introduction

Over the last decade, the World Wide Web (WWW) has grown to contain approximately 5.86 billion pages[1] and is used daily by 40% of the population worldwide.[2] The portion of the Web made up of web pages connected via links is known as Web 2.0 or Document Web. One of the most common usages of the WWW is the search for information. Software systems and named search engines are designed to facilitate a *web search* by performing *web search queries* that access the WWW in a systematic way to find relevant information. For example, the Google Web Search is able to process, on average, 40,000 search queries every second.[3] Even though search engines are able to exploit links on the Document Web effectively, they have two major drawbacks: (1) the relation on the Document Web are not typed. Hence, the meaning of relations cannot be exploited, and (2) classical Web information retrieval techniques are not designed to retrieve information collected across web pages as they are designed to return web pages.

To address this gap, Tim Berners-Lee envisioned the Semantic Web (SW) [24], which is as an extension of the Document Web existing WWW able to present data in the form of a globally linked database that can be accessed and processed by smart agents, dubbed the Web 3.0. To achieve this goal, the data needs (1) to follow common formats and protocols and (2) to be machine-understandable and accessible so that agents can process information as efficiently as human operators. The World Wide Web Consortium (W3C) provides the framework for the SW standardization, which includes syntax, formal description of terms and concepts, vocabularies, knowledge representation and query languages (see Chapter 2 for more information).[4] For example, facts are represented as triples using RDF (Resource Description Framework) format.[5]

The materialization of the SW is known as the Linked Data Web or Linked Data, and has evolved from merely 12 Knowledge Bases (KBs) [12] to more than 10,000 datasets.[6] Recent technological progress in hardware development and network infrastructures has led to the collection of large amounts of data in scenarios as diverse as life sciences, social networking,[7] monitoring industrial plants [120], monitoring open SPARQL endpoints [172], implementing the Internet of Things (IoT), and Cloud Computing [41]. As a result, Knowledge Bases such as

[1] https://www.worldwidewebsize.com/
[2] https://www.internetlivestats.com/internet-users/
[3] https://www.internetlivestats.com/google-search-statistics/
[4] https://www.w3.org/
[5] https://www.w3.org/RDF/
[6] http://lodstats.aksw.org/
[7] http://l3s.de/tweetsKB/

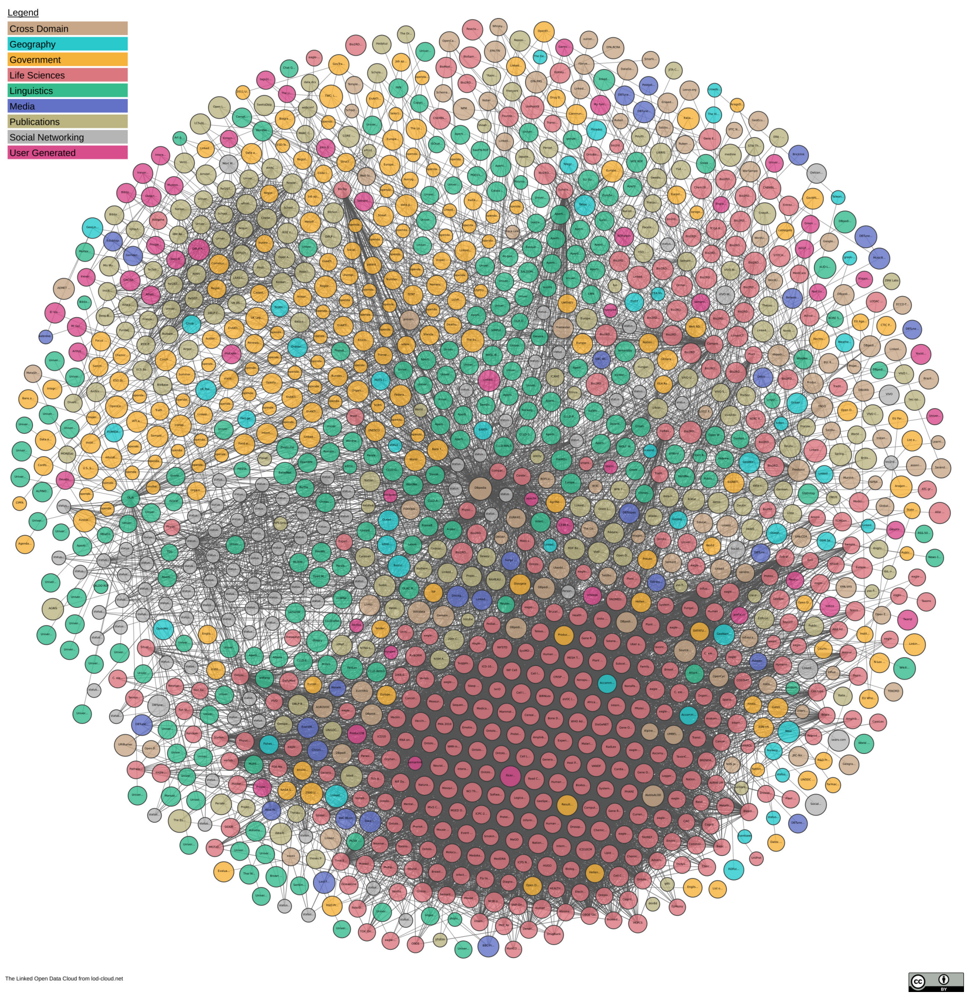


Figure 1.1: The Linked Open Data cloud

DBpedia, the structural representation of Wikipedia pages, includes 4.58 million facts about people, places, species and creative works.[8] The LSQ dataset [172] consists of more than 1.3 billion facts that describe more than 250 million query events on open SPARQL Protocol and RDF Query Language (SPARQL) endpoints. LinkedGeoData includes approximately 1.3 billion triples on geo-spatial entities.[9] The Linked Open Data (LOD) Web is a fraction of the Linked Data Web and consists of Linked Data, which is released under an open license allowing re-use for free.[10] Figure 1.1 depicts the status of the LOD cloud on November 2020.

For data to be published as Linked Data, it must follow a set of practices known as the Linked Data principles (for more information, see Section 2.1): [11]

- *Principle* 1: Use URIs to name things.
- *Principle* 2: Use HTTP URIs to look up those names.
- *Principle* 3: Upon search, provide useful information using standard formats.
- *Principle* 4: Include links to other URIs allowing more things to be discovered.

[8] `https://wiki.dbpedia.org/about`
[9] `http://lodstats.aksw.org/rdfdocs/2267`
[10] `https://en.wikipedia.org/wiki/Linked_data#cite_note-DesignIssues-1`
[11] `http://www.w3.org/DesignIssues/LinkedData.html`

In this work, we focus on the fourth Linked Data principle. The essence of this principle is that given a source set S of RDF resources and a target set T of RDF resources, the aim is to generate new RDF statements that connect resources from S with resources from T. The provision of links between Knowledge Bases is of utmost importance during the creation of Linked Data. Applications such as question answering [187, 204], keyword search [186] and federated query processing engines [173] are dependent on the availability of links on the Web. For example, to answer the query

Example 1.1.
Which targets are involved in blood clotting?

using the biomedical portion of the Question Answering on Linked Data benchmark [205], a question answering framework must utilize links between drugs described in DrugBank and drugs described in Sider to compute complete results.[12]

This constant growth of volume and velocity of KBs has led to an increasing need for linking techniques between resources. There are two main challenges that need to be addressed to allow for the efficient computation of accurate links between Knowledge Bases: accuracy and runtime complexity:

1. Linking approaches need to address the *accuracy challenge*, i.e., they need to generate correct links. A plethora of methods have been developed for this purpose and contain algorithms ranging from genetic programming [33, 85] to probabilistic models [31, 70].

2. Linking frameworks need to address the *challenge of time efficiency*. This challenge is associated with the mere size of knowledge bases that need to be linked. In this work, we focus on the latter challenge with the key motivation of making Semantic Web technologies more amenable to large-scale use, especially (but not only) at industrial scale.

The main focus of this thesis is the implementation of time-efficient and scalable data-linking approaches for knowledge graphs in RDF. Under the declarative representation paradigm, we address the Link Discovery time efficiency challenge by presenting a set of novel approaches that facilitate:

1. the integration of large amounts of data under time or space constraints

2. the scalable execution of complex link specifications to determine candidates for links

3. approaches for linking resources based on temporal or semantic relations that have been paid little attention

1.1 Motivation

Time-efficient LD is of central importance to implementing the vision of the Semantic Web. The baseline approach of computing links between two Knowledge Bases would be a brute force approach. However, this approach would be both time-consuming and computationally expensive, with a time complexity in $O(|S||T|)$ [91, 225], where $|S|$ and $|T|$ are the sizes of the two Knowledge Bases. As a result, a notable amount of previous work on fast and scalable LD has focused on developing approaches that reduce the number of comparisons necessary to compute all pairs of strings with a similarity higher than a predefined threshold, while maintaining the completeness of results. State-of-the-art LD frameworks such as SILK [86] and LIMES [137] rely on string similarities and machine learning to compute links between instances in RDF

[12] `http://qald.aksw.org/index.php?x=task2&q=4`

KBs. While the use of string similarities has been shown to work well in a large number of papers (see, e.g.,[134, 5, 56]), string similarities have the major drawback of not considering the semantics of the sequences of tokens they aim to compare. Hence, most string similarity measures return low scores for pairs of strings such as (`lift`, `elevator`), (`holiday`, `vacation`), (`headmaster`, `principal`) and (`aubergine`, `eggplant`), although they often stand for the same real-world concepts. Edge-counting semantic similarities (e.g., [218, 118, 156]) alleviate this problem by using a dictionary to compute a semantic distance between a sequence of tokens. The synonymy between `aubergine` and `eggplant` would hence lead semantic similarity to assign the pair (`aubergine`, `eggplant`) a similarity score close to 1. The use of semantic similarities has received little attention in LD for at least two reasons: firstly, *semantic similarities scale poorly* and are thus impractical when used on large KBs. Secondly, current works (e.g., [125]) suggest that they lead to no improvement in F-measure.

Event data is increasingly being represented according to the Linked Data principles. The availability of such large collections of event data in the RDF format, as well as the uptake of semantic technologies to represent machine events [164] has created a need to interlink these events. Linking such datasets is essential to support structured machine learning [113, 133, 41, 18] in tasks such as intelligent predictive maintenance for machine data or discovering sequences of query patterns that a triple store is often faced with. Thus, the need for large-scale machine learning on data represented in this format has led to the need for efficient approaches to compute RDF links between resources based on their temporal properties.

While a number of time-efficient approaches for computing links between RDF resources have been developed over the last few years (see [134] for a detailed overview), dedicated approaches for linking resources based on temporal relations have been paid little attention. To the best of our knowledge, only one approach has been developed for computing temporal links between events [191]. The approach presented in [191] is based on the MultiBlock algorithm [86] and employs multi-dimensional blocking to reduce the number of comparisons necessary to compute temporal relations. Throughout detailed evaluation (see Chapter 4 for more details), the experimental results suggest that this approach does not scale to a larger number of events.

While event data is one example of large RDF datasets needing to be linked, other data, such as sensor data, is also generated in the RDF format [83]. Sensor data is used in a plethora of modern applications, including condition monitoring and predictive maintenance in Industry-4.0 applications [25], environmental protection applications, health monitoring systems and traffic monitoring [78]. As a result, sensor data in RDF format facilitates the implementation of condition monitoring and predictive maintenance applications as explainable machine learning solutions, which learn and update OWL axioms periodically to detect (condition monitoring) or predict (predictive maintenance) errors [25]. A key step for learning axioms that generalize well is to learn them across several machines. However, single machines generate independent data streams; hence, time-efficient data integration (in particular LD) approaches must precede machine learning approaches to render machine learning on streams possible. Given that new data batches are available periodically (e.g., every 2 hours), practical applications of machine learning on RDF streams demand scalable data integration solutions that can guarantee complete computation under constraints such as time (i.e., their total runtime for a particular integration task) or expected recall (i.e., the estimated fraction of a given LD task they are guaranteed to complete).

Another approach to improve the scalability of LD frameworks is to use planning algorithms in a manner akin (but not equivalent to) their use in databases [138]. In general, planners rely on cost functions to estimate the runtime of particular portions of Link Specifications. Generally, it has been assumed that this cost function is linear in the planning parameters, i.e., in the size of datasets and the similarity threshold. However, this assumption has never been verified. Additionally, most LD frameworks rely on complex Link Specifications for this purpose. To date, most approaches for improving the execution of Link Specifications have focused on reducing the runtime of the atomic similarity measures used in Link Specifications (see, e.g., [139, 59, 220]). While these algorithms have led to significant runtime improvements, they fail to exploit global knowledge about the Link Specifications to be executed. So far, the execution of Link Specifications has been modeled as a linear process (see [134]), where a Link Specification is commonly rewritten, planned and finally executed. While this architecture has its merits, it fails to use a critical piece of information: *the execution engine knows more about runtimes than the planner once it has executed a portion of the specification.*

The goal of this work is to address the challenge of time-efficient and scalable LD by providing means to:

- improve the execution of single measures (especially temporal and semantic); and
- accelerate the execution of whole specification through planning, which demands the prediction of runtimes.

1.2 Research Questions and Contributions

In this section, we present a set of Research Questions derived from Section 1.1, along with our contributions.

RQ1. How can we accelerate the computation of temporal relations between events?

We address the problem of efficiently computing temporal relations between events by relying on Allen's Interval Algebra [8], as it encompasses all primitive temporal relations between events in Chapter 4. Our approach, dubbed Aegle (Allen's intErval alGebra for Link discovEry), relies on two insights: firstly, the 13 Allen relations can be reduced to 8 simpler relations that all precisely compare either the beginning or end of an event with the beginning or end of another event. Secondly, given that time is ordered, we can reduce the problem of detecting such relations to the problem of finding matching entities in two sorted lists. As this problem has a complexity of $O(n \log n)$, our approach should scale well even for larger datasets. Importantly, our method achieves 100% precision and recall as it computes all temporal relations between events from a source and a target set.

RQ2. Can we scale up the execution of semantic similarities?

To overcome the issue introduced in Section 1.1, we study the effect of semantic similarities on LD, by presenting hECATE, a generic framework for improving the runtime of edge-counting semantic similarities in Chapter 5. The goal of our work is two-fold: first, we present a means to accelerate the computation of four popular bounded edge-counting semantic similarities. We then combine string and semantic similarities using two state-of-the-art machine learning approaches to observe the influence of semantic similarity in LD.

RQ3. How well can we predict the runtime of link specifications?

As mentioned in Section 1.1, the assumption that the runtime cost function is linear has never been verified. Therefore, we study linear, exponential and mixed models for runtime estimation in Chapter 6. The contributions of our work are thus as follows: (1) We present three different models for runtime approximation in planning for LD. (2) We compare these models on six different datasets and study how well they can approximate runtimes of specifications and how well they generalize across datasets. (3) We integrate the models with a state-of-the-art planner for LD as described in [138] and compare their performance using 500 Link Specifications.

RQ4. How well can we compute links under time constraints?

We address the problem of completing the task of link computation under time constraints by proposing Liger (Link discovery with Guaranteed Expected Recall), an approach for LD with partial recall. Liger serves as the first *partial-recall LD approach*. Given a Link Specification L to be executed, Liger aims to efficiently compute a portion of the links returned by L, while achieving a guaranteed expected recall (Chapter 7). Our approach relies on a refinement operator, which allows the exploration of potential solutions to this problem. The main contributions of our work are thus as follows: (1) We present the formal definition of a downward refinement operator that allows the detection of subsumed Link Specifications with partial recall. Moreover, we study its characteristics and provide a novel and efficient approach for partial-recall LD. (2) We use a monotonicity assumption to extend this algorithm so as to make it more time-efficient. (3) We provide both a qualitative and quantitative evaluation of our approach using four benchmark datasets, as well as three new datasets based on real data. In addition to an intrinsic evaluation, we also provide an extrinsic evaluation to quantify the effect of partial-recall LD on machine learning.

RQ5. Can dynamic planning decrease the runtime of link specifications?

We study the execution of Link Specifications in a non-linear process, by proposing and implementing a dynamic flow of information between the engine and the planner so that a LD framework will execute a Link Specification faster. Our approach, Condor makes use of a minute but significant change in the planning and execution of Link Specifications (Chapter 8). The core idea behind our work is to use information generated by the execution engine at runtime to re-evaluate the plans generated by the planner. To this end, we introduce an architectural change to LD frameworks by enabling a flow of information from the execution engine back to the planner. While this change might appear negligible, it has a significant effect on the performance of LD systems, as shown by our evaluation. The contributions of this work are hence as follows: (1) We propose the first planner for link specification that is able to re-plan steps of an input Link Specification L based on the outcome of partial executions of L. By virtue of this behavior, we dub Condor a *dynamic planner*. (2) In addition to being dynamic, Condor goes beyond the state of the art by ensuring that duplicated steps are executed exactly once. Moreover, our planner can also make use of subsumptions between result sets to further reuse previous results of the execution engine. (3) We evaluate our approach on 700 Link Specifications and 7 datasets and show that we significantly outperfom the state of the art.

1.3 Thesis Outline

In this section, we describe the structure of the thesis, which includes 9 chapters. Chapter 1 serves as the introduction of the thesis, and it includes the motivation, research questions, hypotheses and contributions of the author. Chapter 2 introduces the basic notation and formalization used throughout the rest of the thesis. Chapter 3 is dedicated to the state of the art related to our proposed approaches.

Chapters 4–8 contain the main contributions of this thesis. Each chapter includes one approach proposed to deal with another aspect of the challenge of fast and scalable Link Discovery. The said chapters adopt the following structure:

- a *preliminary* section that introduces additional necessary definitions to understand the rest of the chapter, if necessary.
- an *approach* section that describes the methodology behind the proposed method.
- an *evaluation* section in which the approach is evaluated against other approaches that represented the state of the art at the time of writing.

Chapters 4–5 introduce a set of approaches aiming to optimize the efficiency of atomic similarities for Link Discovery. Chapter 4 focuses on our approach for fast execution of time relations in Link Discovery. Chapter 5 introduces an approach towards the efficient computation of four semantic string similarities.

Chapters 6–8 introduce a set of proposed approaches for the fast execution of complex similarities and Link Specifications. Chapter 6 presents an evaluation of runtime models for runtime approximation for Link Discovery. Chapter 7 introduces the first partial-recall Link Discovery algorithm that operates under time constraints. Chapter 8 describes the first dynamic planning algorithm for Link Discovery.

Finally, in Chapter 9, we conclude our thesis and present future extensions for our approaches.

2 Preliminaries

In this Chapter, we introduce the basic definitions and notations that will be used throughout this work. We begin by giving an overview of the Linked Data Paradigm in Section 2.1. Then, we continue with a definition of Link Discovery (LD) in Section 2.2, an overview of declarative LD in Section 2.3, and the characteristics of a Link Specification (LS) and its means of execution in Section 2.4.

2.1 Linked Data

The Linked Data Web is a portion of the SW which often serves as an example to illustrate the principles that should govern a Web of Data. Tim Berners-Lee introduced the "best practices", known as the Linked Data principles, which must be followed for data to be published. These principles are:[1]

- *Principle* 1: Use URIs to name things.
- *Principle* 2: Use HTTP URIs to look up those names.
- *Principle* 3: Upon searching, provide useful information using standard formats.
- *Principle* 4: Including links to other URIs allows more things to be discovered.

The rationale behind the first three Linked Data principles is that in order to publish data on the Web, the "resources" included in the data must be identified using an HTTP Uniform Resource Identifier (URI) [76]. By using the Web architecture term "resource", we focus on things of interest of a particular domain that can refer to both real-world and abstract entities and concepts. An HTTP URI follows the structure:

$$[\mathit{scheme}:][//\mathit{authority}][\mathit{path}][?\mathit{query}][\#fragment]$$

The *authority* tag is divided into three subcomponents:

$$\mathit{authority} = [userinfo@]\mathit{host}[:\mathit{port}]$$

For example, the Wikipedia article about the "Adventures and Campaigns" within the "Dungeon and Dragons" setting looks like:

[1] `http://www.w3.org/DesignIssues/LinkedData.html`

Example 2.1.
https://en.wikipedia.org/wiki/Dungeons_And_Dragons#Adventures_and_campaigns

The usage of HTTP URIs is preferred for two main reasons: (1) they are simple so that any domain owner can create globally unique names, and (2) they can be used to provide access to the information describing the identified entity [76]. As a result, an HTTP URI should be de-referencable so that an HTTP client can search the URI using the HTTP protocol and retrieve a description of the resource that the URI describes [12]. Note that for disambiguation reasons, different URIs are used to identify a real-world or abstract object and the document that describes the said object. The description of a resource can be represented either using HTML, if the end-user is human, or using the Resource Description Framework (RDF) if the end-user is a machine.

The RDF [99] data model is a W3C specification used to represent knowledge on the Web in a generic, standardized manner. In RDF, information is represented as statements, called RDF triples. A triple consists of a subject, a predicate and an object, mimicking the structure of a simple sentence.

Definition 2.2 (RDF Triple)**.** *Given the pairwise disjoint infinite sets of IRIs ($\mathcal{I}$), of blank nodes ($\mathcal{B}$), and RDF literals ($\mathcal{L}$), a triple* $(c, p, o) \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$ *is called an RDF triple, where c is the subject, p the predicate and o the object.*

The subject of a triple denotes the resource and it can either be an Internationalized Resource Identifier (IRI) or an unnamed entity (blank node). An IRI is a generalization of URIs allowing the use of UNICODE. The predicate of an RDF triple describes the type of relation between the subject and the object and it is allowed to be identified using an IRI. Finally, the object of the triple can either be an IRI, a blank node or an RDF literal in the form of a UNICODE string. The following triple serves as an example and states that *Steven Erikson is the author of "Malazan: Book of the Fallen"*:

Example 2.3.
subject c: $< http://dbpedia.org/Steven\_Erikson >$
predicate p: $< http://dbpedia.org/author >$
object o: $< http://dbpedia.org/Malazan\_Book\_of\_the\_Fallen >$

In order to express more complex knowledge using RDF triples, a number of languages are built on top of RDF and extend it with more expressive semantics. One of these languages is the Resource Description Framework Schema (RDFS), which includes a set of classes with certain properties using the RDF data model. It provides RDF vocabularies as shared terminology, intended to structure RDF resources. Another expressive representation language based on formal logic is Web Ontology Language (OWL). OWL allows logical reasoning to be performed on the knowledge, and enables access to knowledge which is only implicitly modeled [79]. An example of an OWL term is $< http://www.w3.org/2002/07/owl\#sameAs >$. For simplicity reasons, it can be re-written as $owl : sameAs$, where we substitute the namespace $http://www.w3.org/2002/07/owl\#$ with the prefix *owl*. Both RDFS and OWL follow W3C standardization.

If we combine a set of RDF triples, we have an RDF Graph.

Definition 2.4 (RDF Graph)**.** *An RDF Graph is a finite set of RDF triples.*

An RDF graph itself represents a resource, which is located at a certain location on the Web and thus has an associated IRI - the graph IRI [12]. In order to visualize an RDF Graph, we assign IRIs representing the subject and object of a set of triples as the graph nodes, and the predicate as the directed edges connecting the subject and object of a triple.

Definition 2.5 (RDF Knowledge Base). *An RDF Knowledge Base (KB) is a finite set of RDF Graphs.*

To publish an RDF Graph or a KB on the Web, we need means for serializations. Since the RDF model is a data model and not a data format, there have been a set of serializations following the W3C standards, such as RDF/XML [21], Turtle [20], N-Triples and RDF/JSON.

With the uptake in Semantic Web technologies, a vast amount of data is transformed using the RDF model described previously, following the first three Linked Data principles. Since large collection of RDF data is available, there is a need to link those datasets following the 4th Linked Data principle: "Including links to other URIs allows more things to be discovered".

The types of links described by the 4th Linked Data principle can be classified into two main categories:

1. relationship links, where the objective is to find and link resources, also known as Link Discovery (LD), and
2. vocabulary links, where the goal is to find and link definitions of vocabulary terms, such as classes and properties, also known as Ontology Matching (OM).

In this work, we focus on the first type of links: relationship links. Therefore, we use the term "relationship link" and "link" interchangeably throughout this work.

2.2 Link Discovery

Given a set of source RDF resources S and a set of target RDF resources T, the goal of LD is to identify RDF statements that connect the resources from S with resources from T. Formally:

Definition 2.6 (Link Discovery). *Given two sets of RDF resources S and T from two (not necessarily distinct) KBs, as well as a relation Rel, the main goal of LD is to discover the* mapping $\mathcal{M}$ *as the set* $\{(s,t) \in S \times T : Rel(s,t)\}$.

A relation example can be `owl:sameAs`, where the LD goal is to identify the set of pairs of resources that represent the same entities. In that case, the LD task is equivalent to task deduplication [101] and entity matching [102]. In this work, we use the term "Link Discovery" as a hypernym for record linkage, deduplication, entity resolution and entity matching. Another example of a relation is the bf relation, where LD aims to find the set of source events (as RDF resources) that began and ended before a set of target events (as RDF resources).

The identification of links between resources is of utmost importance in applications such as question answering [187], condition monitoring and predictive maintenance in Industry-4.0 applications [25], data intergration [22], environmental protection applications, health monitoring systems[2] and traffic monitoring [78].

2.3 Declarative Link Discovery

Over the years, several solutions have been proposed that address the identification of links between resources. One approach is to manually detect the set of pairs (s, t) with $s \in S, t \in T$ that satisfy the relation Rel. However, the Linked Open Data cloud has evolved from merely 12 KBs [12] into more than $10,000$ KBs. [3] As a result, the manual approach towards LD is an unscalable and time consuming task.

[2]`https://www.healthcare.siemens.co.uk/magnetic-resonance-imaging/options-and-upgrades/clinical-applications/interactive-realtime-imaging`

[3]`http://lodstats.aksw.org/`

Therefore, declarative LD frameworks focus on identifying a set of links, $\mathcal{M}^* \subseteq S \times T$, whose similarity $m(s,t)$ is equal to or greater than threshold θ. More formally:

Definition 2.7 (Declarative Link Discovery)**.** *Given two sets of RDF resources S and T from two (not necessarily distinct) KBs, a* (complex) similarity function $m : S \times T \to [0,1]$*, and a* similarity threshold $\theta \in [0,1]$*, the main goal of the declarative LD is to compute the* mapping $\mathcal{M}^*$ *as the set* $\{(s,t) \in S \times T : m(s,t) \geq \theta\}$*.*

Additionally, using a *distance function* σ, $\mathcal{M}^*$ can be computed as $\{(s,t) \in S \times T : \sigma(s,t) \leq \varpi\}$, where $\varpi \in [0,\infty)$ is a *distance threshold.* Note that a distance function σ can be transformed into a similarity function m by setting $\sigma(s,t) = \frac{1}{1+m(s,t)}$. The distance threshold ϖ can be transformed into a similarity threshold θ by setting $\theta = \frac{1}{1+\varpi}$. As a result, distance and similarity functions are used interchangeably within this work.

A similarity function m is *atomic* iff it consists of exactly one similarity measure used on a particular set of properties i.e. $p_s, p_t \in \mathcal{P}$, with $\mathcal{P}$ being the set of all properties. We write $m(s,t,p_s,p_t)$, to signify the similarity of s and t w.r.t. their properties p_s resp. p_t, or $m(p_s,p_t)$ for simplicity. It can also be *complex* if it is a combination of two similarity functions using a *metric operator* (e.g., max, min).

Declarative LD is one of the families of solutions towards fulfilling the 4th Linked Data principle. Other approaches include (but are not limited to): statistical models [28], probabilistic models using Conditional Random Fields (CRF) [107], Machine Learning (ML) classifiers [122] and inference algorithms [170]. However, in this work we focus on declarative Link Discovery, therefore the terms declarative Link Discovery and Link Discovery are used interchangeably throughout the rest of this manuscript. In this work, we only consider declarative LD frameworks as they are the most commonly used. Other types of frameworks are described in [134]

2.4 Link Specification

Given that $\mathcal{M}$ is generally difficult to compute directly, some declarative LD frameworks compute the approximation $\mathcal{M}^*$ using Link Specifications (LSs). A LS provides the means to express the requirements under which the relation *Rel* holds. Several grammars have been used for describing a LS in previous works [86].

Based on these grammars, a LS can be divided into two categories:

1. *atomic*, where the LS is described as a pair $L = (m(p_s,p_t),\theta)$ with $\theta \in [0,1] \cup \{\varnothing\}$. We denote the empty specification with $L_\emptyset$. Additionally, we define a *filter* as a pair (f,ζ), where f is either empty (denoted ϵ), a similarity measure or a combination of similarity measures and ζ is a threshold. Note that an atomic LS can be regarded as a filter and be written as a triple (f,ζ,X) with $X = S \times T$.

2. *complex*, where the LS is described as a triple $L = (f,\zeta,op(L_1,L_2))$. For $L = (f,\zeta,\omega(L_1,L_2))$, we call $\omega = op(L)$ *the operator* of L. We call (f,ζ) the *filter of* L and denote it with $\varphi(L)$. We call L_1 the *left sub-specification* or *left child* and L_2 the *right sub-specification* or *right child* of L. In this work, we limit ourselves to formalizing the specification operators $\sqcap$, $\sqcup$, $\setminus$ (difference), since the remaining operators (e.g., $\oplus$ (exclusive or)) can be derived from this subset.

The *size of* L, denoted $|L|$, is defined as follows: If L is atomic, then $|L| = 1$. For complex LS $L = (f,\zeta,\omega(L_1,L_2))$, we set $L = |L_1| + |L_2| + 1$. We denote the semantics (i.e., the results of a LS for given sets of resources S and T) of a LS L by $[[L]]$ and call it a *mapping.* Note that the mapping of an empty LS is $[[L]] = \emptyset$. The semantics of LSs are then as shown in Table 2.1.

Table 2.1: Semantics of Link Specifications

L	$[[L]]$
$L_\emptyset$	$\emptyset$
$(m(p_s,p_t),\theta)$	$\begin{cases}\{(s,t)\in S\times T \wedge m(s,t,p_s,p_t)\geq\theta\} \text{ if } \theta\in[0,1]\\ \emptyset \text{ else.}\end{cases}$
(f,ζ,X)	$\begin{cases}\{(s,t)\in[[X]] : m(s,t)\geq\zeta\} \text{ if } f=\epsilon\\ \{(s,t)\in[[X]] : f(s,t)\geq\zeta\} \text{ else.}\end{cases}$
$\sqcap(L_1,L_2)$	$\{(s,t)\|(s,t)\in[[L_1]]\wedge(s,t)\in[[L_2]]\}$
$\sqcup(L_1,L_2)$	$\{(s,t)\|(s,t)\in[[L_1]]\vee(s,t)\in[[L_2]]\}$
$\backslash(L_1,L_2)$	$\{(s,t)\|(s,t)\in[[L_1]]\wedge(s,t)\notin[[L_2]]\}$

To elucidate the grammar behind a LS, we present an example LS in Figure 2.1. Based on the figure, we notice the following: (1) the operator is $\sqcup$, (2) $\varphi(L) = (\epsilon, 0.60)$, (3) L_1 is $(\textit{trigrams}(\texttt{:title}, \texttt{:name}), 0.30)$, (4) L_2 is $(\textit{cosine}(\texttt{:label}, \texttt{:label}), 0.70)$ and (5) $|L|$ is 3.

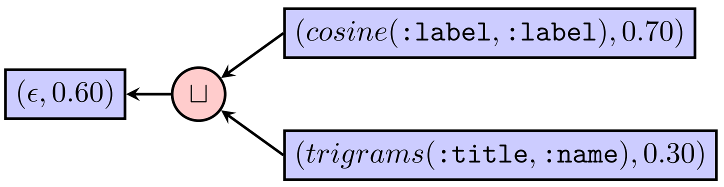

$(\textit{cosine}(\texttt{:label}, \texttt{:label}), 0.70)$
$(\epsilon, 0.60)$
$\sqcup$
$(\textit{trigrams}(\texttt{:title}, \texttt{:name}), 0.30)$

Figure 2.1: Graphical representation of a complex LS

To compute the mapping $[[L]]$ (which corresponds to the output of L for a given pair (S,T)), LD frameworks implement (at least parts of) a generic, linear architecture consisting of an execution engine, an optional rewriter and a planner (see [134] for more details). Figure 2.2 gives an overview of the workflow. The *rewriter* performs algebraic operations to transform the input LS L into a LS L' (with $[[L]] = [[L']]$) that is potentially faster to execute. The most common planner is the *canonical planner* (dubbed Canonical), which simply traverses L in post-order and has its results computed in that order by the execution engine.[4] There are multiple ways to traverse the L, such as pre-order and in-order, however in this work we use the semantics of planners introduced in [138].

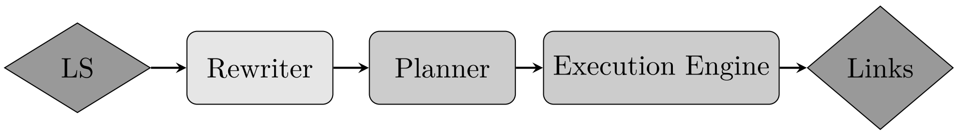


Figure 2.2: Linear architecture of the execution of a LS

For the LS shown in Fig. 2.1, the execution plan returned by Canonical consists of the following steps:

- *Step* 1: Compute the mapping $M_1 = [[(\texttt{trigrams}(:\textit{title}, :\textit{name}), 0.30)]]$ for each pair of resources, whose p_s property `title` and p_t property `name` has a trigrams similarity equal to, or greater than 0.30.

[4]Note that the planner and engine are not necessarily distinct in existing implementations.

- *Step* 2: Compute the mapping $M_2 = [[(\texttt{cosine}(:\mathit{label}, :\mathit{label}), 0.70)]]$ of pairs of resources, whose p_s property `label` and p_t property `label`, has a cosine similarity is equal to, or greater than 0.70.
- *Step* 3: Compute $M_3 = M_1 \sqcup M_2$, while abiding by the semantics described in Table 2.1.
- *Step* 4: Compute M_4 by filtering M_3 and keeping only the pairs that have a similarity equal to, or greater than 0.60.

Given that there is a 1-1 correspondence between a LS and the plan generated by the Canonical planner, we will reuse the representation of a LS devised above for plans. The sequence of steps for such a plan is then to be understood as the sequence of steps that would be derived by Canonical for the LS displayed.

3 Related Work

In this chapter, we present a set of related state-of-the-art approaches towards fast and scalable declarative LD. As we explained in Section 2.3, some declarative LD frameworks utilize LSs to compute an approximation of $\mathcal{M}$, $\mathcal{M}^* = \{(s,t) \in S \times T : m(s,t) \geq \theta\}$. As we explained in Section 2.4, in order to express more complex relations, many frameworks use metric operators (e.g. max, min) to combine similarity functions. Therefore, we divide this chapter into two parts: (1) related state-of-the-art towards the runtime optimization of atomic similarities such as string similarities (Section 3.1 for syntactic string similarities and Section 3.3 for semantic string similarities), and time relations (Section 3.2); and (2) related state-of-the-art towards the runtime optimization of complex similarities focusing on fast execution of LSs (Section 3.4) and declarative LD frameworks and tools (Section 3.5).

3.1 A Systematic Survey of String Similarity Joins for Link Discovery

Preamble This section is based on the paper "Systematic Survey on String Similarity Joins for Link Discovery by Kleanthi Georgala and Axel-Cyrille Ngonga Ngomo", which is currently submitted in a peer-reviewed journal and is under review. It is the first systematic survey for String Similarity Joins (SSJs) for LD that covers the time period 2008-2018. The author designed the methodology, presented and evaluated the algorithms presented therein, and co-wrote the paper.

3.1.1 Motivation

A large number of Link Discovery (LD) frameworks have been developed to address the need for links in and across datasets (see [134] for a survey). Most of the existing LD frameworks rely on combining atomic similarity and distance functions (e.g., the Levenshtein distance, the Jaccard similarity) to complex similarity or distance functions. These complex functions are then applied to property values of pairs (s,t) of RDF resources to identify links between these resources. Since the most frequent data type in RDF datasets is the "string" data type [134], the majority of LD tools utilize string similarities (amongst others) to carry out comparisons between pairs of string attributes. In this work, we hence survey the use of string similarities and distances in LD.

String similarities can be divided into two main categories:

1. *syntactic string similarities*, which determine the similarity of two strings based exclusively on the sequence of characters the strings are made of; and

2. *semantic string similarities*, which aim to determine the similarity of two strings based on the likeness of the meaning of said strings.

In both cases, LD frameworks use string similarities to compare attribute values of pairs of resources $(s, t) \in S \times T$ with a threshold θ. The similarity score of a pair of strings being higher or lower than said threshold is then used further in the process of suggesting the existence of a link between the two resources $(s, t) \in S \times T$ [5, 56]. In the following, we will say that the property values of two resources *match* if their similarity is higher (resp. lower) than θ. By extension, we will also say that two resources match if their property values match.

As mention in Section 1.1, the baseline approach for computing matches is a brute-force approach, which iterates over every pair $(s, t) \in S \times T$ to assess whether the attribute values of s and t match. Since this approach is infeasible, a notable amount of previous work on fast and scalable LD has focused on developing approaches that reduce the number of comparisons necessary to compute all pairs of strings with a similarity higher than a predefined threshold θ, while maintaining the completeness of results, i.e., while ensuring that they return all matching pairs of resources (s, t). These methods include but are not limited to: blocking [159, 197], clustering [227, 175], filtering [179, 33], hashing [70, 193], execution optimization approaches [65, 138] and string similarity joins (SSJs) [221, 222].

Numerous approaches and evaluations regarding the aforementioned approaches and their application to identifying links between resources [53, 171, 196, 104, 146] have been published in the literature. However, to the best of our knowledge, there has been no systematic survey for SSJs and their application to LD. We address this research gap by surveying and presenting SSJs that were published between 2008 and 2018. We focus on syntactic string similarity joins for two reasons: first, previous works ([125]) suggest that semantic string similarities do not improve the F-measure of the task of matching resources. Secondly, our findings (Section 3.1.3) suggest that there have been a very small number of works pertaining to the use of semantic SSJs in LD. [1]

Based on the literature, we classify SSJ approaches into two main categories: (1) filter-verification approaches; and (2) tree-based approaches. We give an overview of the reported performance of these approaches based on the corresponding papers and address their ability to produce accurate links in a time-efficient manner. The results we present are not only of importance for LD but also for its related fields, including record linkage, deduplication and instance matching.

3.1.2 Preliminaries

We begin by presenting the necessary notations regarding strings, string similarities and string similarity joins. Then, we describe the relation between LD and SSJs.

Definition 3.1 (String). *A string $r \in \Sigma^*$ is a finite sequence of characters over an alphabet Σ. The length of a string r is denoted by $|r|$.*

Definition 3.2 (String Similarity Measure). *A string similarity measure (or simply string similarity) $\mu : \Sigma^* \times \Sigma^* \rightarrow \mathbb{R}^+$ maps a pair of strings $(r, g) \in \Sigma^* \times \Sigma^*$ to a similarity value. String similarities are often used against a similarity threshold $\lambda \in \mathbb{R}^+$. If $\mu(r, g) \geq \lambda$, then r and g are considered a match.*

[1]For an extensive analysis of semantic string similarities and their application to LD, see Chapter 5.

Definition 3.3 (String Similarity Metric). *A string similarity metric (or simply string metric) $d(r, g)$, where $r, g \in \Sigma^*$ is a function such that $d : \Sigma^* \times \Sigma^* \to \mathbb{R}^{+}$*[2]*, where Σ^* is the set of all possible strings over Σ. A string metric quantifies the distance between two strings. Given a distance threshold $\gamma \in \mathbb{R}^+$, $d(r, g) > \gamma$ implies that r, g are not considered a match.*

Like in the literature [134], we refer to both string similarity measures and string similarity metrics as string similarities. String similarities have been studied and surveyed extensively over the years [174, 160, 49, 63, 154]. They can be divided into 3 main categories:

1. *Character-based similarities* quantify the similarity between two strings r and g by means of the number of single-character transformations (e.g., deletions, insertions, substitutions, transpositions) required to transform g into r. An often-used family of character-based string similarities are the edit distances, of which the Levenshtein distance is the most commonly used [114]. Another edit distance metric is the Damerau-Levenshtein distance [42], which includes the transposition of two adjacent characters to the set of possible operations between strings. The Jaro [1], Jaro-Winkler [217] and Hamming distances [153] are other character-based metrics used extensively in linking [52].

2. *Token-based similarities* (also called *term-based*)transform input strings into (ordered) sets of tokens and use the similarity between the two sets of tokens as a proxy for the similarity between the two initial strings. The computation of the set of tokens is often carried out by (1) tokenizing the string using special characters (e.g., white spaces) or by (2) using a sliding window of size q over the string ($q - grams$). Some well-known and widely-used methods include (but are not limited to), the overlap similarity, the Jaccard similarity [88], the cosine similarity, Dice's coefficient [48] and the Manhattan distance [106].

3. *Hybrid similarities*, combine the advantages of character and token-based similarities. For example, the Monge-Elkan similarity measure [129] begins by tokenizing the input strings. Then, it calculates the Levenshtein or the Jaro-Winkler distance for each pair of tokens, and computes the similarity of the strings as the average similarity score over all pairs of tokens. SoftTFIDF [211], the generalized edit similarity (GES) and fuzzy overlap [213] are further hybrid string similarities found in the literature.

Definition 3.4 (String Similarity Join). *Given two sets of strings $R \subseteq \Sigma^*$ and $G \subseteq \Sigma^*$, a string measure μ (or string metric d) and similarity threshold $\lambda \in \mathbb{R}^+$ (or a metric threshold $\gamma \in \mathbb{R}^+$), the goal of a string similarity join (SSJ) is to find the set of string pairs $(r, g) \in R \times G$ such that $\{(r, g) \in R \times G \mid \mu(r, g) \geq \lambda\}$ (resp. $\{(r, g) \in R \times G \mid d(r, g) \leq \gamma\}$).*

In this section, we focus on optimization methods for the execution of atomic similarity functions, $m(s, t, p_s, p_t)$. Thus, for a particular set of string properties p_s, p_t of s and t resp., the problem of finding the set of pairs (s, t) with $m(s, t, p_s, p_t) \geq \theta$ becomes equivalent to the task of finding the set of pairs (s, t) which, mapped to values $p_s(s)$ resp. $p_t(t)$, satisfy the SSJ condition $\mu(p_s(s), p_t(t)) \geq \theta$. Consequently, efficient execution of SSJ is of central importance in LD. In the following, we present a survey of SSJs for LD.

3.1.3 Systematic Survey Methodology

Our systematic survey of the literature on SSJs for LD is based on the approaches presented in [98, 128]. First, we present our methodology for finding surveys related to our survey. Then, we introduce our systematic survey methodology for identifying SSJs for LD.

[2] $\mathbb{R}^+ = \{x \in \mathbb{R} | x \geq 0\}$. This notation holds for the rest of this work.

Related Surveys

In this section, we introduce our methodology for discovering related surveys to SSJs for LD.

Question Formulation The questions we aimed to answer by conducting a systematic search for related surveys are the following:

- $RS - Q_1$: What is the current status of systematic surveys for SSJ in LD?
- $RS - Q_2$: On which topics do related surveys for LD focus?
- $RS - Q_3$: What type of surveys have been conducted for SSJs?

Eligibility criteria We created two lists of inclusion/exclusion criteria for related surveys. The papers had to abide by all inclusion criteria and by none of the exclusion criteria to be part of our survey:

- Inclusion Criteria
  - Work published in English between 2008 and 2018.
  - Surveys on algorithms and frameworks that improve the performance of string similarities for LD.
  - Surveys on string similarity join approaches for LD.
- Exclusion criteria
  - Works that were not peer-reviewed or published.
  - Works that were published as poster abstracts.

Search Strategy We defined a set of keywords related to our survey. These were: study, survey, evaluation, string similarity, string similarity join, link discovery, record linkage, deduplication, entity resolution and entity matching. We used those keywords to derive the following query:

- (`intitle:` *survey* OR `intitle:` *study* OR `intitle:` *evaluation*) AND (*string similarity* OR *string similarity joins*) AND (*link discovery* OR *record linkage* OR *deduplication* OR *entity resolution* OR *entity matching*)

The `intitle` tag was used to specify that the keywords *survey*, *study* and *evaluation* should be present in the title of the paper, where applicable.

Keyword searches were carried out using the following list of search engines, digital libraries, journals and conferences:

- Google Scholar [3]
- IEEE Xplore Digital Library (IEEE Xplore) [4]
- ACM DL [5]
- Science Direct [6]

[3]`http://scholar.google.com/`
[4]`https://ieeexplore.ieee.org/Xplore/home.jsp`
[5]`http://dl.acm.org/`
[6]`http://www.sciencedirect.com/`

- Semantic Web Journal (SWJ) [7]
- Journal of Web Semantics (JWS) [8]
- Journal of Data and Knowledge Engineering (JDWE) [9]

Search Methodology Steps To conduct our systematic literature search of related surveys, we applied a six-phase search methodology:

1. Apply keywords to the search engine using the time frame 2008-2018.
2. Scan article titles based on inclusion/exclusion criteria.
3. Import the output from step 2 to a reference manager software to remove duplicates. Here, we used Mendeley 1.19.3[10] as it is free and has the required functionality for deduplication.
4. Review abstracts according to include/exclude criteria.
5. Read through papers, looking for approaches that fit the inclusion criteria and excluded papers that fit the exclusion criteria.
6. Retrieve the corresponding papers from references included in the papers of step 5 or papers that cited the papers of step 5 that fit the inclusion/exclusion criteria.

Table 3.1 provides an overview of the number of papers retrieved in each phase of the search methodology.

Table 3.1: Number of retrieved related surveys for each phase of the search methodology

Search	Steps					
Engines	1	2	3	4	5	6
Google Scholar	99	42	41	29	25	34
IEEE Xplore	5	0	0	0	0	0
ACM DL	1	1	0	0	0	0
Science Direct	38	3	2	2	2	5
SWJ	34	3	2	0	0	0
JWS	38	3	0	0	0	0
JDWE	38	3	0	0	0	0

Core Survey

Now that we have introduced our approach for identifying surveys related to our work, we continue by presenting our core methodology for the literature review of SSJs in LD.

[7] `http://www.semantic-web-journal.net/`
[8] `http://www.websemanticsjournal.org/`
[9] `http://www.journals.elsevier.com/data-and-knowledge-engineering/`
[10] `http://www.mendeley.com/`

Question Formulation We begin by formalizing the research questions we aimed to answer in the evaluation section of our survey:

$CS - Q_1$: Can SSJs be divided into categories (e.g., based on a common implementation template?) What are the characteristics of each category?

$CS - Q_2$: What challenges are associated with SSJs?

$CS - Q_3$: What trends can be observed across SSJ approaches over the survey time frame?

Eligibility criteria We created two lists of inclusion/exclusion criteria for related publications. The papers had to abide by all inclusion criteria and by none of the exclusion criteria to be part of our survey:

- Inclusion Criteria
  - Work published in English between 2008 and 2018.
  - String similarity join approaches for LD.
  - (String) Set similarity join approaches for LD.
- Exclusion criteria
  - Work that was not peer-reviewed or published.
  - Work that was published as a poster abstract.

Search strategy Based on the research questions and the eligibility criteria, we defined the following set of related keywords: string similarity, string similarity join, link discovery, record linkage, deduplication, entity resolution and entity matching. We used those keywords to create the following query:

- (*string similarity* OR *string similarity joins*) AND (*link discovery* OR *record linkage* OR *deduplication* OR *entity resolution* OR *entity matching*)

The query defined by this keyword search was applied to the list of search engines, digital libraries, journals and conferences shown in Section 3.1.3.

Search Methodology Steps To conduct our systematic survey of SSJs for LD we applied a seven-phase search methodology:

1. Apply keywords to the search engine using the time frame 2008-2018.
2. Scan article titles based on inclusion/exclusion criteria.
3. Import output from step 2 to a reference manager software to remove duplicates. Here, we used Mendeley 1.19.3[11] as it is free and has functionality for deduplication.
4. Review abstracts according to include/exclude criteria.
5. Read through papers, looking for approaches that fit the inclusion criteria and exclude papers that fit the exclusion criteria.
6. Retrieve new papers from references that cite the papers of step 5.

[11] `http://www.mendeley.com/`

7. Scan references from the survey papers that passed steps 5 and 6 of the related surveys methodology from Section 3.1.3 and retrieve new papers that fulfilled the inclusion/exclusion criteria.

Additionally, any new survey that was discovered during our search methodology steps to identify SSJ papers for LD, were added to the results of the related surveys. Table 3.2 provides an overview of the number of retrieved papers through each phase of the search methodology. Due to the large amount of results obtained from Google Scholar, step 2 was applied to the top-1000 relevant results.

Table 3.2: Number of retrieved related surveys for each phase of the search methodology

Search	Steps						
Engines	1	2	3	4	5	6	7
Google Scholar	2,290	339	318	223	20	18	2
IEEE Xplore	10	9	3	2	0	0	0
ACM DL	13	9	0	0	0	0	0
Science Direct	59	20	15	10	0	0	0
SWJ	28	15	14	6	0	0	0
JWS	59	20	0	0	0	0	0
JDWE	59	20	0	0	0	0	0

3.1.4 Related Survey Results

In this section, we present the results of our systematic search of related surveys for SSJs in LD from Section 3.1.3. We divide our related work section into two subsections that cover the basic topics of our research: link discovery and string similarity joins. We begin by answering $RS-Q_1$ from Section 3.1.3 by stating that during our systematic search for applications of SSJ to LD, we did not come across any survey (systematic or not), study or evaluation focusing on that topic, for the time period 2008-2018.

Link Discovery Surveys

Conducting surveys and empirical studies on LD frameworks and systems has drawn a lot of attention in the scientific community over the past decade. To answer $RS{-}Q_2$ from Section 3.1.3, the studies presented as related work for LD are divided into two categories: (1) comparison of LD frameworks or approaches, where the comparison criteria include (but are not limited to) efficiency, and (2) comparison of specific runtime improvement approaches. The surveys that belong to the first category give an overview of the existing LD tools or techniques, including their ability to reduce the search space and improve efficiency, as part of a wider comparison schema. The second category of papers includes surveys on blocking, indexing, clustering and genetic programming that optimize the runtime of the linking process. Our work differs from the first category because we focus only on the challenge of efficiency in LD, and we differentiate ourselves from the second category, by specifically focusing on SSJs to reduce the number of comparisons between strings, and thus, the time complexity.

A detailed comparison and evaluation of some interlinking tools that utilize LOD datasets can be found in [134]. In that survey, the authors compared 11 state-of-the-art LD frameworks including *LIMES* [135], *Silk* [210], *KnoFuss* [147], *Zhishi.links* [149] and *AgreementMaker* [38]. The comparison of tools was made based on a generic LD architecture and the criteria included effectiveness, low configuration, and tuning effort, (apart from efficiency). In a similar manner, [77] discusses different algorithms for identifying links between ontology instances based on their ability to handle large-scale datasets. Another related work regarding the evaluation of interlinking tools for the Web of Data was published in [158]. This paper serves as an empirical study on three LD frameworks, whose performance was evaluated by human experts based on the number of links, number of matched records and common terms. Additionally, based on the finding of [158], the authors conducted a case study on interconnecting an e-Learning dataset to DBpedia [157]. [131] presents a different comparison approach, by focusing on various architectural features of the systems, such as linking techniques, domains, interface characteristics and input/output format types.

In a similar manner, there have been many surveys about different machine learning-based approaches for LD. [54] provides a comparative study of various unsupervised classification techniques, such as Support Vector Machines (SVMs), Decision Trees and Bayes classifiers. [196] provides an extensive comparison of ten supervised approaches for LD on six different datasets, evaluating both efficiency and effectiveness. The authors of [163] compare various supervised and unsupervised classifiers for historical census linkage.

In the field of benchmarking Instance Matching (IM) tools, [43] offers a thorough study on the current state-of-the-art IM benchmarks for Semantic Web data, based on reproducibility of results, availability and equity among systems. One of the most prominent benchmarks in IM is the *Ontology Alignment Evaluation Initiative (OAEI)* [60].[12] OAEI provides a two-fold evaluation: (1) evaluation conducted on real-data to measure the performance of tools under real linking scenarios, and (2) evaluation using synthetic data for better understanding of the advantages and disadvantages of each system when presented with heterogeneous datasets. Another rising benchmark, LANCE [13] (Linked Data instance matching Benchmark Generator) provides a domain independent structure with the ability to link any provided linked dataset, supporting complex semantics-aware test cases [178].

Efficiency and scalability in LD has been studied thoroughly in many related publications. Surveys such as [47, 35, 105, 6] provide an overview of the existing methods and approaches to improve the linking runtime. Some of those methods include blocking [159, 197, 152, 86, 50, 198], clustering [53, 227, 175], adaptive windows [224, 126], filtering [179, 33], hashing [70, 193], genetic programming [85], and rule-based approaches [3, 110].

String Similarity Join Surveys

Due to their numerous applications, SSJs have been the center of attention in many previous studies. In [91], the authors presented and evaluated 14 SSJs on a variety of datasets, dividing their experiments into the following categories: (1) experiments on large datasets, (2) experiments on small datasets, (3) scalability experiments, and (4) self-joins ($R = G$) In a similar manner, [225] studies a set of SSJs, providing a detailed analysis and categorization. Furthermore, there are four more surveys, [190, 14, 182, 7], that focus on SSJs based on MapReduce [45] for further improving the time complexity and scalability of some SSJs. To answer $RS - Q_3$ from Section 3.1.3, the aforementioned surveys for SSJs either do not present a comparison or evaluation of the different SSJs, or they serve as an experimental evaluation on SSJs rather than a systematic survey.

During our search for related work regarding SSJs for LD, we came across papers that study fields relevant to SSJs, but differ in the task at hand. One example is a String Similarity Search (SSS), where a given a set of strings $R \subseteq \Sigma^*$ and a query string $z \subseteq \Sigma^*$, a string measure μ (or

[12]`http://oaei.ontologymatching.org/`
[13]`http://www.ics.forth.gr/isl/lance/`

string metric d) and similarity threshold $\lambda \in \mathbb{R}^+$ (or a metric threshold $\tau \in \mathbb{R}^+$), the goal of an SSS is to find the set of strings such that $\{r \in R \mid \mu(r, z) \geq \lambda\}$ (resp. $\{r \in R \mid d(r, z) \leq \gamma\}$). [225, 212] are examples of such surveys. However, even though SSSs and SSJs are closely related, often studied together and many algorithms are implemented to perform both, SSSs are beyond the scope of this survey. Such surveys include: [123, 89, 62]. Note that, algorithms that implement SSJs or Set Similarity Joins and SSSs, and satisfy the eligibility criteria of our survey are included in our work, such as ***IndexGram*** [155], ***IndexChunk*** [155], ***Para-Join*** [90].

3.1.5 String Similarity Joins

The following presents the results of our systematic survey on SSJs for LD implemented as laid out in Section 3.1.3. To answer $CS - Q_1$, we divide existing SSJs for LD into two basic categories: (1) filter-verify approaches, and (2) tree-based approaches. Tables 3.3- 3.12 provide an overview of the characteristics of the filter-verify and the tree-based approaches, presented in the following sections. The filter-verify approaches are distributed among different tables (Tables 3.3- 3.11). We divide the SSJs in multiple tables for readability purposes. The order in which each SSJ appears in a table is dictated by its corresponding order in the text. We also present and discuss the parallel version of some SSJ approaches along with supplementary characteristics. Note that we use the notation introduced in Chapter 2 and Section 3.1 instead of the symbols used in the publications we surveyed for the sake of improved understandability and comparability. Also note that several publications on SSJs use the term "edit-distance" to mean different distances. For the sake of clarity, we use the name of the intended distance (e.g., Levenshtein distance) instead of "edit-distance".

Filter-verify approaches

The filter-verify approach consists of two steps: a *filtering step* and a *verification step.* Filtering aims to produce a set of candidate pairs for the verification step by pruning non-matches, thus reducing the number of comparisons carried out in the verification step. The first step in the generation of candidate pairs consists of mapping the input strings to signatures. The algorithm then aims to pair strings together that share common signatures. The pairs are filtered thereafter and those pairs that pass the filtering stage are regarded as candidate pairs. Finally, in the verification step, each candidate pair is verified by computing its real similarity value.

Filter-verify approaches can be divided into three categories based on how they generate signatures: (1) *prefix-based* (Tables 3.3- 3.6), (2) *partition-based* (Tables 3.7- 3.10) and (3) *neighborhood-based* (Table 3.11). Before we present SSJs that fall into these categories, we give an overview of the filters commonly used across the three categories of approaches aforementioned.

Overview of Filters The approaches that implement these filters use $q - grams$ to produce the set of signatures for each string, unless specified otherwise. In the following equations, τ is the Levenshtein distance threshold and δ is the overlap similarity threshold.

First, we define a *positional* $q - gram$ of a string r as a pair $(i, r[i \dots i + q - 1]$, where $r[i \dots i+q-1]$ is the $q-gram$ of r that starts at position i, counting on the extended string [75]. The set W_r of all positional $q - grams$ of a string r is the set of all $|r| + q - 1$ pairs constructed from all $q - grams$ of r [75].

- Count filtering [75, 176, 115]: here, the underlying assumption is that if two strings r and g have a distance less than or equal to γ, or their similarity is equal to or greater than λ, then their signatures must share at least C common signatures. Examples of means to compute C are as follows:

$$C = \begin{cases} |r| + 1 - q - (q\tau) & \text{for Levenshtein distance} \\ \delta & \text{for overlap} \\ \lceil \delta |r| \rceil & \text{for jaccard} \\ \lceil \delta^2 |r| \rceil & \text{for cosine} \\ \left\lceil \frac{\delta}{2-\delta} |r| \right\rceil & \text{for dice.} \end{cases} \tag{3.1}$$

- Length filtering [75]: here, the underlying assumption is that if two strings r and g have a distance less than or equal to γ, or their similarity is equal to or greater than λ, then the following must hold for their lengths:

$$\begin{aligned} |r| - \tau \leq |g| \leq |r| + \tau, & \quad \text{for Levenshtein distance} \\ \delta|r| \leq |g| \leq \frac{|r|}{\delta}, & \quad \text{for jaccard} \\ \delta^2|r| \leq |g| \leq \frac{|r|}{\delta^2}, & \quad \text{for cosine} \\ \frac{\delta}{2-\delta}|r| \leq |g| \leq \frac{2-\delta}{\delta}|r|, & \quad \text{for dice} \end{aligned} \tag{3.2}$$

- Positional filtering [75][14]: here, the underlying assumption is that if two strings r and g have a distance less than or equal to γ, or their similarity is equal to or greater than λ, then a positional $q-gram$ in r cannot correspond to a positional $q-gram$ of g that differs from it by more than å positions. For the Levenshtein distance, å $= \tau$ [75] and for the overlap similarity, å $= \delta$ [221].

- Prefix filtering [30]: assume a global ordering that orders all tokens in the sets of signatures. Based on [30], a global ordering eliminates higher frequency elements from the prefix filtering and thereby expects to minimize the number of comparisons.[15] If two strings, r and g have a distance less than or equal to γ, or their similarity is equal to or greater than λ, then $pre_r \cap pre_g \neq \emptyset$, where pre_r and pre_g are the prefixes (set of the first v signatures) of the r and g respectively. Examples of means to compute v are as follows:

$$v = \begin{cases} q\tau + 1 & \text{for the Levenshtein distance} \\ |r| - \delta + 1 & \text{for overlap} \\ \lfloor (1-\delta)|r|) \rfloor + 1 & \text{for jaccard} \\ \lfloor (1-\delta^2)|r|) \rfloor + 1 & \text{for cosine} \\ \left\lfloor (1 - \frac{\delta}{2-\delta})|r| \right\rfloor + 1 & \text{for dice.} \end{cases} \tag{3.3}$$

Prefix-based approaches Prefix-based approaches build an inverted index, where the signatures serve as the keys, and a list of strings that contain the corresponding signatures serve as the values.[16] Remember that the main idea behind prefix-based approaches is that if two

[14]In bibliography, "position" and "positional" filtering are used interchangeably.

[15]A global ordering is a descending or ascending order applied to all tokens of all strings, based on e.g. inverse document frequency (*idf*), alphabetical order, token frequency (*tf*).

[16]The list can be either ordered or not ordered, depending on the approach

strings r and g have a distance less than or equal to γ, or their similarity is equal to or greater than λ, then they must share a subset of common prefixes, where a prefix is the set of the first v signatures based on a global ordering.

For each prefix-based SSJ, we present the following:

- the string similarity(ies) supported,
- the algorithm(s) implemented to produce the signatures,
- the type(s) of filtering used,
- the optimization(s) performed at the verification stage (if any), and
- the parameters used in configuration.

We begin our discussion of filter-verify approaches with an SSJ for the Levenshtein distance, ***Ed-Join*** [219]. Remember that τ denotes the threshold for any SSJ which relies on the Levenshtein distance. ***Ed-Join*** introduces the idea of using the information provided by both shared and unshared *q-grams* in the signature list of strings to identify candidate pairs for verification. The ***Ed-Join*** algorithm receives a set of arrays that are ordered based on their length as input. Each array contains a set of $q-grams$ ordered in decreasing order based on their *idf* values. To reduce the size of prefixes needed for comparison, ***Ed-Join*** uses minimum prefixes, where a minimum prefix is the shortest prefix of the $q-gram$ array, such that if all the $q-grams$ in the minimum prefix are mismatched, it will incure at least $\tau+1$ errors [219]. Hence, if all $q-grams$ in the minimum prefix are mismatched, it will incur at least $\tau+1$ edit errors [219]. Based on this idea, the authors proposed a new filter, called location-based mismatch filter, which was used during the candidate pair generation. For the verification stage, ***Ed-Join*** utilizes a set of filters to speed up the comparisons: (1) count and position filtering, (2) location-based mismatch filtering and, (3) the new content-based mismatch filtering. Content-based mismatch filtering serves as a complementary filter to the location-based mismatch filtering, based on the observation that several edit errors actually occur within the same unshared $q-grams$. Additionally, the authors of [219] provide an extension of the ***All-Pairs*** [19] algorithms for the Levenshtein distance, named ***All-Paird-Ed***, which they use as the baseline in their experiments. Finally, to study the effects of each filter individually, they create ***Ed-Join-I*** by removing the content-based filter from ***Ed-Join***.

Ed-Join and especially ***All-Paird-Ed*** have inspired the authors of [215] to create the first chunk-based method for exact Levenshtein similarity join, ***VChunkJoin***. ***VChunkJoin*** proposes a set of tail-restricted Chunk Boundary Dictionary (CBS) schemes, where strings are divided into non-overlapping, disjoint sub-strings (chunks). Their idea is to minimize the space needed to create an index and store the signatures. The algorithm uses the location-based mismatch filtering and content-based mismatch filtering introduced in [219], but also introduces two new filters: chunk number filtering and virtual CBD filtering. The first new filter aims to number the signatures associated with a string, and the second stores the resulting chunk numbers rather than storing and indexing the resulting chunks.

Our next filter-verify SSJ algorithm is an approach that combines both *p*ositional and *p*refix filtering, dubbed ***PPJoin*** [221]. ***PPJoin*** improves the efficiency of various string similarities such as cosine, jaccard, overlap, Hamming distance and Levenshtein distance, along with their weighted versions. The ***PPJoin*** algorithm takes as input a record multiset that is ordered based on length; each set contains a set of prefixes ordered by ascending order based on their document frequency (*df*) values. The authors of [221] propose two novel techniques for optimizing the effectiveness and efficiency of the filtering phase: first, they store both the tokens and their position in the prefix set; then they minimize the index size using an upper bound based on the

Table 3.3: Characteristics of prefix-based String Similarity Joins. The ♣ symbol indicates that the approach also implements a parallel version of its algorithm.

Approach	**String Similarities**	**Signatures**	**Filtering Techniques**	**Verification Optimization**	**Parameters**	**Year**
Ed-Join [219]	Levenshtein distance	$q-grams$, ordering by *idf*, minimum prefix	location-based mismatch filtering	count+positional filtering and location-based mismatch filtering and content-based mismatch filtering	τ, q	2008
Ed-Join-I [219]	Levenshtein distance	$q-grams$, ordering by *idf*, minimum prefix	location-based mismatch filtering	count+positional filtering and location-based mismatch filtering	τ, q	2008
All-Pairs-Ed [219]	Levenshtein distance	$q-grams$, ordering by *idf*	prefix filtering	count+positional filtering	τ, q	2008
VChunkJoin [215]	Levenshtein similarity	tail-restricted Chunk Boundary Dictionary schemes (vchunks), ordering by *idf*	chunk number filtering or virtual CBD filtering, and location-based mismatch filtering and prefix+ length filtering	content-based mismatch filtering and [203]	τ, q	2012
PPJoin [221]	jaccard cosine overlap Levenshtein distance Hamming distance	tokenization, ordering by *df*, prefix length constraints, length ordering	prefix and positional filtering	comparison of suffixes	λ or τ	2008
PPJoin+ [221]	jaccard cosine, overlap Levenshtein distance Hamming distance	tokenization, ordering by *df*, prefix length constraints, length ordering	prefix and positional filtering	suffix filtering	λ or τ	2008
MPJoin♣ [162]	jaccard cosine dice	$q-grams$, ordering by feature frequency, use overlap and set size bound	min-prefix filtering (prefix filtering)	merge-join algorithm, comparison of suffixes	q, λ	2011
Adapt-Join [214]	overlap dice cosine jaccard Levenshtein distance	variable-length prefix scheme using a cost model, use of delta inverted indexes	prefix filtering	-	l, λ or τ	2012

position of a token in the prefix set (positional filtering). For the comparison phase, ***PPJoin***'s verification algorithm compares the last tokens of both prefixes, considering that only the suffix of the smaller token needs to be intersected with the entire other record [221]. As a further improvement to the verification step, the authors of ***PPJoin*** propose a novel suffix filtering method (***PPJoin+***), as a generalization of prefix filtering to work with suffixes. Therefore, any candidate pair that passes the filtering stage needs to be further verified before the actual comparison happens.

Table 3.4: Characteristics of prefix-based String Similarity Joins. The † symbol indicates that the method is available as a parallel processing algorithm only. The ♣ symbol indicates that the approach also implements a parallel version of its algorithm.

Approach	String Similarities	Signatures	Filtering Techniques	Verification Optimization	Parameters	Year
P4Join♣ [180]	Tanimoto	same as ***PPJoin*** for bit-vectors	same as ***PPJoin*** for bit-vectors: length,prefix and positional filtering	-	λ	2015
PairWiseMR† [55]	Okapi BM25	stemming, stop-words removal, df-cut	MapReduce's mapper	MapReduce's reducer	df -cut thresholds	2008
EFS-S† [209]	jaccard, Tanimoto cosine	Basic Token Ordering (BTO) or One-Phase Token Ordering (OPTO)	prefix, length and positional filtering with Basic Kernel (BK) or Indexed Kernel (PK) with prefix, positional filtering	Basic Record Join (BSJ) or One-Phase Record Join (OPRJ)	number of nodes, λ	2010
SSJ-2† [17]	cosine	same as [209], partition in buckets	prefix filtering with load balancing techniques	same as [55], and compute document similarity once	λ	2010
SSJ-2R† [17]	cosine	same as [209] partition in buckets, partitioning of remainder file in chunks	prefix filtering with load balancing techniques	same as [55], and compute document similarity once, avoid remote random access	λ, number of chunks K	2010
IndexGram [155]	Levenshtein similarity	$q-grams$, prefix filtering with lower bound	naive count or matching or error estimation-based filtering	-	τ	2011
IndexChunk [155]	Levenshtein similarity	$q-chunks$, prefix filtering with lower bound	naive count or matching or error estimation-based filtering	-	τ	2011

PPJoin has become a state-of-the-art approach for optimizing the runtime of many string similarities. Hence, a series of methods have built upon and extended the ***PPJoin*** framework to solve variants of the similarity join problem. One such approach is ***MPJoin*** [162], which studies the problem of Set Similarity Joins. Set Similarity Joins serve as a subcategory of SSJs by utilizing only token-based string similarities. ***MPJoin*** focuses on minimizing the computational costs of identifying candidate pairs by introducing a generalization of prefix filtering, the min-prefix filtering method. ***PPJoin*** has been extended in [180] to handle Privacy Preserving Record Linkage (PPRL). The authors of [180] introduce ***P4Join***, an alternative version of ***PPJoin*** that efficiently compares bit vectors for PPRL. The algorithm uses the set bit positions (indexes) as tokens, so that the length of a record is the number of set bits in its bit array. Then, it alters the filters used in ***PPJoin*** to fit the encrypted data. To improve the runtime even further, they introduce a GPU-based version of ***P4Join*** and a hybrid version that uses both GPUs and CPUs.

Adapt-Join [214] is another approach that utilizes prefix filtering. The main difference between existing approaches and ***Adapt-Join*** is that it proposes an adaptive selection of choosing a prefix scheme, named the variable-length prefix scheme. The algorithm dynamically selects the length(s) of the prefix(es) of each object, using a cost-based model. To do so, the authors of [214] use delta inverse indexes to handle the adaptive nature of their algorithm.

The authors of ***PairWiseMR*** [55] propose a pairwise similarity approach based on MapReduce to optimize the runtime of comparisons between documents in large collections. First, the algorithm converts each text into a bag-of-words using stemming. Then, it associates each term with document ids that include this term, using a standard inverted index. The mapper emits the term as the key, uses a tuple consisting of the document id and term weight as the value, groups together the tuples and writes them to the disk (postings). Before the verification stage, the ***PairWiseMR*** algorithm uses *df-cut* pruning, where the fraction of the terms with the highest document frequency is excluded. Then, the MapReduce creates $\frac{1}{2}w(w-1)$ tuples with document ids as keys, where w is the average length of a posting. Finally, the reducer sums up all the individual scores from each pair and generates the final similarity between a pair of documents.

EFS-S, another approach that utilizes MapReduce for time efficiency, is presented in [209]. ***EFS-S*** implements parallel set similarity joins for both self-join and $R \neq G$. For the self-join scenario, the authors of [209] propose two global orderings of the tokens: (1) Basic Token Ordering (BTO), where the tokens are ordered based on frequency by taking advantage of the "map", "reduce" and "combine" functions of MapReduce; and (2) One-Phase Token Ordering (OPTO), which is similar to BTO but the tokens are ordered in memory instead. Once the prefix sets are produced, the algorithm extracts the prefix tokens from each record and distributes the document ids (RID) and the join-attribute value pairs to the reducers. The reducers compute the similarities of the join-attributes and then output the RIDs of matched pairs. The distribution is performed either by using individual tokens, where each token is a key, or by grouping tokens together, where an artificial key is created by combining multiple keys. The identification of matched RIDs is performed by using a nested loop approach to compute the similarity of the join-attribute values (Basic Kernal - BK) or by using ***PPJoin+*** [221] (Indexed Kernel - PK). Once each reducer has identified the RIDs of matched pairs, the algorithm proceeds in combining the individual values into an aggregated result. This is achieved either by Basic Record Join (BRJ), where the list of RID pairs is provided as input to the map functions, or by One-Phase Record Join (OPRJ), where the list is sent to all mappers before the input data is consumed by the mapper. For the $R \neq G$, the authors used the same ordering methods, with the further step of excluding any token of G that was not found in R. Additionally, for the identification of RIDs of matched pairs at each mapper and for the final stage of pairs aggregation, the approach utilized both same methods as in the self-join case, with small alternations to fit the $R \neq G$ join. Finally, the authors of [209] propose two alternatives of the main algorithm, for cases where there is no sufficient memory available to store all data.

Based on the implementation optimizations introduced in ***PairWiseMR*** [55] and ***EFS-S*** [209], the authors of [17] propose two new approaches for document similarity self-join. First, ***SSJ-2*** utilizes prefix filtering and extends [55] by introducing a load balancing technique for partitioning the data into the MapReduce mappers. In the verification phase, it also improves the best approach introduced in [209], by allowing the similarity between two documents to be computed only once, even if they appear in more than one mapper. The authors also introduce ***SSJ-2R*** as an extension of ***SSJ-2***, which avoids sending the full collection of documents to every MapReduce node multiple times.

Two other approaches focusing on Levenshtein similarity join are proposed in [155]. The authors of [155] introduce two methods, ***IndexGram*** and ***IndexChunk***, which differ in the

Table 3.5: Characteristics of prefix-based String Similarity Joins. The † symbol indicates that the method is available only as a parallel processing algorithm. The ♣ symbol indicates that the approach also implements a parallel version of its algorithm. The ♢ symbol refers to the additional characteristics of the parallel version.

Approach	String Similarities	Signatures	Filtering Techniques	Verification Optimization	Parameters	Year
Fast-Join [213]	fuzzy dice, fuzzy cosine, fuzzy jaccard	tokenization, removal of maximum number of largest signatures (c-1) tokens	token-sensitive filtering	construction of a weighted bi-graph with loose constraints	λ, η	2011
Partition-NED [213]	Levenshtein distance	same as [216]	Minimal-Edit-Distance or Duplication Pruning	construction of a weighted bi- graph with loose constraints	τ, η	2011
V-SMART-Join† [127]	any similarity with Shuffling Invariant Property	convert data into multisets, represent multisets as tuples	-	Online-Aggregation or Look-Up or Sharding algorithm	τ or λ	2012
MGJoin♣ [168]	cosine jaccard dice	tokenization, multiple global orderings: random selection or random selection with reverse order TF_1 or TF_2	multiple prefix filtering	multiple global orderings♢	λ, number of orderings	2013
REEDED [194]	weighted Levenshtein distance and other distances of the edit-distance family	tokenization	length-aware, character-aware filtering	dynamic programming	τ	2013
SN-Join [121]	full expansion, selective expansion	compute signatures off-line and choose best online based based on 2 estimators	prefix filtering	-	λ	2013
SI-Join [121]	full expansion, selective expansion	compute signatures off-line and choose best online based based on 2 estimators	prefix, length filtering	-	λ	2013
SS-J [40]	Levenshtein distance	$q-grams$	position, length, count filtering	-	τ	2014

way they produce signatures for strings. ***IndexGram*** uses $q-grams$ to extract signatures, whereas ***IndexChunk*** uses $q-chunks$. $q-chunks$ are substrings of length q, that start at $1+(iq)$-th position of the string. Their idea of using an asymmetric scheme for signatures was based on the intuition there is no “one fits all” approach for generating signatures among different datasets. To minimize the size of the $q - grams$ and $q - chunks$, they proposed a lower bound on common signatures by using prefix and count filtering. Finally, they introduced a new set of filters to help minimize the number of comparisons between candidates. These are: (1) naive count filtering, (2) matching filtering, which transforms candidate matches into a bi-partie graph that prunes the least amount of vertices so the final graph does not violate a set of constraints; and (3) error estimation-based filtering.

Our next algorithm, ***Fast-Join*** [213] proposes a new set of string similarities, named fuzzy similarities, which serve as alternatives to existing string measures and metrics to improve the effectiveness and efficiency of SSJs. ***Fast-Join*** introduces an extension of prefix filtering for token-based similarities, named the token-sensitive scheme, and an extension of the ***Partition-ED*** [216] method, named ***Partition-NED***, for character-based string metrics. For the verification phase, the algorithm performs a comparison of tokens by constructing a weighted bi-graph, using an upper bound of the maximal weight by relaxing the matching condition.

V-SMART-Join [127] belongs to the MapReduce framework category for SSJs. This approach accommodates similarities between sets, multisets and vectors using any similarity that has the Shuffling Invariant Property (SIP), where the order of elements is not known. The algorithm begins with the *joining phase*, which transforms the data into multisets; each multiset is represented using tuples, a tuple for each element of the alphabet that belongs to the multiset. Therefore, the SSJ problem is transformed into finding the set of multiset pairs whose similarity is above a predefined similarity threshold. The authors of [127] proposed three different *joining* algorithms: (1)the Online-Aggregation algorithm, (2) the Look-up algorithm, and (3) the Sharding algorithm, a hybrid between the first two techniques. The second and final step of the algorithm consists of the similarity phase, which computes the similarity between candidate pairs, utilizing the functionality of the MapReduce framework.

MGJoin [168] introduces a multi-prefix filtering scheme for fast SSJs. The authors of [168] propose the idea of using a set of different global orderings, which are applied to the input data to minimize the number of candidate pairs for verification. First, the algorithm uses one global ordering to create the signature index. Then, for each string and its similarity candidate, it compares their prefixes generated by another global ordering. ***MGJoin*** introduces three global orderings based on permutations in the token universe: (1) random selection, (2) random selection with reverse ordering, and (3) term-frequency (TF) ordering as global ordering followed by (a) a reverse TF ordering(TF_1), or (b) a randomly selected global ordering (TF_2). ***MGJoin*** has also been extended to incorporate the MapReduce framework (***RIDPairsMGJoin***), which utilizes the multi-filtering scheme in the verification stage. Finally, the authors of [168] propose a hybrid method between ***PPJoin+*** and ***MGJoin***, extending the ***PPJoin+*** approach to utilize multi-prefix filtering.

REEDED [194] extends the ***Pass-Join*** framework to handle weighted edit-distances. ***REEDED*** utilizes a length-aware filter, followed by a character-aware filter, to generate the set of candidate pairs. At the verification stage, a dynamic programming implementation is used to guarantee the fast computation of similarities.

The authors of [121] propose an innovative idea of implementing an expansion-based framework for SSJs. Their main idea is to expand the token set of a string with applicable synonyms. To do so, they propose two expansion methods: (1) full expansion (FE) that uses all possible synonyms; and (2) selective expansion (SE), which uses a greedy algorithm for the selection of applicable synonyms, with guarantees pertaining to optimality under certain conditions. The first algorithm, ***SN-Join*** performs prefix filtering using the smallest $\lceil(1-\lambda)|r|\rceil$ tokens, based on a global ordering for any string r. Then it performs comparisons between the pairs that passed the filtering stage. The second algorithm, ***SI-Join*** extends ***SN-Join*** by applying length filtering along with prefix filtering. For comparison reasons, they replaced both pruning methods with Locality-Sensitive Hashing (LSH) filters to study their effects in an expansion-based system. To optimize the candidate pairs generation stage, they proposed the calculation of a variety of signatures off-line, and then selected the best method on-line, based on two different estimators. The authors of [121] propose four different signature schemes: (1) ITF1, which sorts the tokens from data tokens and synonyms separately by decreasing *itf*, and then arranges the tokens from tables first, followed by those from synonyms; (2) ITF2 that separately sorts the tokens and synonyms, but arranges the synonyms first followed by the tables; (3) ITF3 that sorts all tokens together in decreasing *itf* order; and (4) ITF4 that generates all possibilities of expanded sets for each string, then sorts the tokens by decreasing *itf* order in all the expanded sets.

Our next approach presents two methods towards continuous SSJs, where the data is represented as dynamic streams, in contrast to the previous batch-oriented approaches. Based on [40], a data stream $\mathbb{S} = \{(str_1, ts_1), (str_2, ts_2) \ldots\}$ is composed by a series of tuples in the form of (str_i, ts_i), where str_i represents a string and ts_i is its corresponding timestamp. The authors of [40] propose two methods: (1) ***SS-J*** and (2) ***RSS-J***. ***SS-J*** incorporates count, length and position filtering, whereas ***RSS-J*** utilizes the two latter filters combined with count filtering designed for asymmetric signatures [155]. The verification stage of both methods is based on the idea of a *sliding window.* A *sliding window* preserves the constant capacity of a time interval or the number of tuples from a data stream. Any newly arrived tuple is inserted into the *sliding window* and an expired tuple gets evicted according to its capacity. For ***SS-J***, the verification is performed between tuples of the recent window and tuples of the older windows. The result of each basic window is an answer set grouped by the older basic window id. For ***RSS-J***, computing the similarity between two strings r, g is allowed if the following holds: $-\frac{(\tau-(|r|-|g|)}{2} \leq i - j \leq \frac{(\tau+(|r|-|g|)}{2}, \forall i \leq |s|, j \leq |g|$, where i, j are character indexes of r, g resp. (coordinates based verification).

We continue our discussion with ***ClusterJoin*** [177], which provides an SSJ framework for every metric distance.[17] The ***ClusterJoin*** algorithm consists of three steps: first, the data is partitioned to allow parallel processing with MapReduce. To overcome the data skewness drawback, the authors of [177] implement a non-uniform space-partitioning-based approach by sampling the data and allowing the sampled set to determine the space partitioning. Then, the algorithm samples the data to choose *anchor points* as the centers of the partitions (clusters). Additionally, it uses a set of filters to determine whether or not *query points* need to be verified against the *anchor points.* The algorithm splits large partitions by using 2D-hashing, to ensure even-load balancing between partitions. Merging is also allowed if there is a huge overlap between partitions. The authors of [177] provide a generic filtering prototype for the distance metric, along with a set of filters for a specific set of metrics (e.g. Euclidean distance, total variation distance, 1-Norm distance and Hamming distance). The second step of ***ClusterJoin*** is named "the mapping phase", where the remaining points are mapped to a partition for verification. The final phase is the verification stage, where each machine performs comparisons between the data points of each partition in parallel.

The authors of [51] introduce an approach for the efficient execution of the Jaro-Winkler distance, named ***BJaWink***. To minimize the comparisons between strings, ***BJaWink*** includes a set of length-based, range-based and character-based filters. For the latter, they developed a trie structure to avoid character-index lookups for every pair of strings. Additionally, they present a parallel approach to the original algorithm based on a thread-pool-based technique.

PSH [92] is another SSJ approach that focuses solely on the Damerau-Levenshtein distance. The algorithm begins by generating bitwise signatures for each string using 32-bit unsigned integers. Each string is transformed into a sequence of 1s and 0s that represent the set of integer

[17]`https://en.wikipedia.org/wiki/Metric_(mathematics)`

Table 3.6: Characteristics of prefix-based String Similarity Joins. The ♣ symbol indicates that the corresponding approach implements a parallel version of its algorithm. The ♢ symbol refers to the additional characteristics of the parallel version.

Approach	**String Similarities**	**Signatures**	**Filtering Techniques**	**Verification Optimization**	**Parameters**	**Year**
RSS-J [40]	Levenshtein distance	improved version of [155]	position, length, count filtering	coordinates verification	τ	2014
ClusterJoin [177]	any metric distance	vector space model	specific filter per metric	parallel verification of each partition	τ	2014
BJaWink♣ [51]	Jaro-Winkler distance	ordered permutation of a word based on any consistent total ordering of letters	length and ranged and character-based filtering	-	τ, number of CPUs♢	2014
PSH [92]	Damerau-Levenshtein distance	bitwise signatures using 32-bit unsigned integers	signature filtering and naive or probabilistic hashing	Hash Neighborhood Generation	τ	2016
MFKC+♣ [207]	MFKC [181]	k most frequent characters filtering	first frequency hash intersection	-	λ, k number of CPUs♢	2017
SSPS [183]	any string similarity	prefix and suffix of strings	prefix and length and positional and count filtering	-	λ or τ	2017
PBIJoin [169]	dice, jaccard, cosine	tokenization, prefixes ordered based on *tf* or alphabetical, partition-based inverted index	prefix filtering	-	λ	2017
SAIJoin [169]	dice, jaccard, cosine	tokenization, prefixes ordered based on *tf* or alphabetical, similarity-aware inverted index	prefix filtering	-	λ	2017

numbers (0 to 9), followed by the latin alphabet. Then, the signature filtering compresses the signatures into a compact primitive data format, and a hashing (naive or probabilistic) method indexes the mapping of signatures and strings into hash buckets. Finally, for the verification stage, a function using bitwise operators is used to calculate the bucket codes that may contain approximate matches for a string.

We continue our discussion with ***MFKC+*** [207], an approach that focuses on improving the runtime of MFKC [181], a novel string-distance function based on most frequent k characters. Each string is associated with a hash map, where the characters are the keys and the frequency of each character is the value, ordered by frequency in descending order. The algorithm keeps the most frequent k characters as signatures for each string. To minimize the comparisons of strings, the ***MFKC+*** algorithm utilizes a set of filters: (1) first frequency filtering, and (2) hash intersection filtering. A pair of strings that succeeds in both filters is then verified by computing the real MFKC similarity.

Our next approach, ***SSPS*** [183] introduces the idea of generating signatures by using both the prefix and the suffix of a string. Then using a series of filters (count, length and positional), it identifies the set of candidate pairs that will be sent for comparison. ***SSPS*** utilizes MapReduce at each stage of the filter-verification framework to minimize time complexity.

Our next publication implements and tests two different indexing approaches to support different similarity thresholds. [169] presents two indexing approaches: (1) Partition-based inverted index (***PBIJoin***) that selects some representative values within the threshold interval and builds incremental inverted indices; and (2) a similarity-aware index (***SAIJoin***) that utilizes a threshold upper bound for a token to be selected as a prefix token.

Partition-based approaches Partition-based approaches perform a global ordering on the strings and then partition them into sets of disjoint segments. After the non-overlapping partitions are generated, the algorithm utilizes a *substring selection* method that chooses the sub-sets of segments that will be used to identify the matched pairs for verification. The partition-based approaches are based on the idea that if g and r are matched against a threshold, then g must contain a substring that matches a segment of r [117].

For each partition-based SSJ, we study the following characteristics:

- the string similarity(ies) supported,
- the algorithm(s) implemented to produce the signatures,
- the type(s) of substring selection implemented,
- the type(s) of filtering used,
- the optimization(s) performed at the verification stage (if any), and
- the parameters used in configuration.

Pass-Join [117] presents a partition-based method for solving the efficiency problem of string similarities. The main algorithm starts by ordering a set of strings by length and alphabetical order. Then, it partitions each string into $\tau + 1$ disjoint segments, with length greater than 1. For each segment, ***Pass-Join*** builds inverted indices, and for each string r, it selects a set of its substrings. Then it searches for the selected substrings within the inverted indices. If a selected substring appears in the inverted index, each string g on the inverted list of this substring is considered a possible match for r. For each string, ***Pass-Join*** finds a set of matching strings using 4 different substring selection techniques that satisfy the completeness criterion: (1) a length-based method that selects all sub-strings of particular lengths as signatures for a string; (2) a shift-based method that extends the length-based method by considering the position of the segments; (3) a position-aware method that extends the previous two selection methods by limiting the minimum and maximum start positions of sub-strings for signatures; and (4) a multi-match-aware method that discards a subset of matching sub-strings incase two strings have multiple matching segments. Based on the experiments carried out in [117], the multi-match-aware substring selection is the technique with the lowest time complexity. Additionally, the authors of [117] presented two optimizations for verification: (1) a length-aware verification, where they used the length difference to estimate the minimum number of edit operations; and (2) an extension-based verification that partitions strings into 3 parts: matching, left and right, and prunes a pair if the left and right parts do not match.

The same research group has published a set of extensions for ***Pass-Join***. Beginning with ***SegFilter*** [116], this approach focuses on optimizing the normalized Levenshtein distance and introduces iterative-aware verification. In contrast to [117], [116] supports $R \neq G$. Futhermore, ***Para-Join*** [90] extends the partition-based framework by implementing two pruning techniques: (1) content filtering, which is similar to content-based mismatch filtering but introduces a stronger filter condition, and (2) an effective indexing strategy that indexes longer strings and chooses sub-strings from shorter strings to find candidate pairs based on the indexes of longer strings. Additionally, ***Para-Join*** offers a parallel version of its original algorithm to speed up the similarity joins.

[119] proposes 3 extensions of the ***Pass-Join*** algorithm. First, ***Pass-JoinK*** follows exactly the same steps as ***Pass-Join***, but instead partitions the strings into $\tau + K$ disjoint segments. Secondly, ***Pass-JoinKMR*** combines ***Pass-JoinK*** with MapReduce and uses the extension-based verification method for comparisons. Thirdly, ***Pass-JoinKMRS*** combines ***Pass-JoinK*** with MapReduce and uses the length-aware verification method for comparisons. On a similar note, [223] introduces two methods of parallelizing ***Pass-Join***. The first is ***ParaLL-Join***,[18] which splits the alphabet into partitions (*joint-tokens*) using a Z-Collapse algorithm.[19] For each string, it calculates the joint frequency vector, and for each joint-token, it calculates the range of the frequency distribution needed to split the data into segments. ***ParaLL-Join*** implements the same verification process as in [117] using a multi-threading technique. The second is ***Pada-Join***, which introduces a parallel processing version of ***Pass-Join*** in distributed systems using Spark.[20]

Bi-Filtering JOIN [81] is another method that serves as an extension of ***Pass-Join***. It proposes a partition approach utilizing a bi-directional filtering that generates the same candidate pairs, regardless of the direction of filtering. As it is stated in the paper, *the forward filtering uses segments of short strings to select the sub-strings of current string, and, the backward filtering uses the segments of long strings to select the sub-strings of current string* [81]. The intersection of the two aforementioned filters is the set of candidate pairs. The selection of substrings is carried out using a re-design multi-match-aware selection method [117] and the comparison of pairs is performed using the extension-based verification method [117].

The authors of [94] propose three partition-based methods, inspired by ***Pass-Join***. The first method ***TISM***, introduces the idea of partitioning the target dataset G into clusters of strings with equal lengths (length filtering). Then, the algorithm partitions each cluster into blocks, using an *exemplar* string as the representative of each block. Each string $g \in G$ is partitioned into $\tau + 1$ segments, whereas each string of the source dataset, $r \in R$, is partitioned into segments of a particular size. Then, for each source string, ***TISM*** identifies the subset of strings of G with whom it shares at least one common segment. After that, the algorithm uses length filtering to determine the candidate pairs for verification. The second method, ***SM***, does not partition the clusters into blocks but rather assigns a set of strings as candidate pairs based on length filtering and overlapping segments. The third method ***SFM*** enhances the previous method by introducing the character's frequency filtering. Finally, for verification, all methods use dynamic programming to speed up the comparisons.

Another publication proposed a further extension of ***Pass-Join***, named ***MassJoin*** [46], specifically targeting token-based SSJs. Based on the idea that the number of partitions for token-based string similarities is not straightforward and fixed, ***MassJoin*** generates signatures for the source and target datasets using different strategies. Each set of source tokens r is partitioned using an even partition scheme into $U + 1$ segments of similar length, dictated by a tight upper bound, with U being a function of $|r|$ and the string similarity threshold. Each set of target tokens g is partitioned using a combination of the position-aware and multi-match-

[18]The original name of the algorithm was ***Para-Join***, but has been renamed to avoid confusion with [90]

[19]`https://en.wikipedia.org/wiki/Wave_function_collapse`

[20]`https://spark.apache.org/`

aware method, which guarantees no false negatives. To reduce the number of key-value pairs generated by MapReduce, the authors of [46] propose a novel merge method, which decreases the complexity of the task from $O(|r|^3)$ to $O(|r|)$ for a single set of source tokens, without loosening the pruning power of the approach. Also, to reduce the transmission cost of candidate pairs from the filtering phase to verification, ***MassJoin*** includes a *light-weight filter unit* to replace the original strings and speed up computations.

An extention of ***MassJoin*** is introduced in [199], named ***MLS-Join***, which improves its *Merge+Light* technique for self-joins. The ***MLS-Join*** algorithm starts by partitioning the strings in $\tau + 1$ non-overlapping segments, then splits each segment into characters. The algorithm creates a vector that includes the frequency of each character based on a fixed character order. Due to space limitations, the authors group the characters into n sets and keep the vector with length of n. ***MLS-Join*** uses the multi-match-aware select substring scheme [117], which eliminates duplicated signatures generated by different segments. For comparison, the algorithm uses the length-aware verification method [117] and scans the input dataset only once.

We continue our discussion with ***Landmark-Join*** [132], an approach that introduces a *hash-join* method that partitions the strings into buckets to achieve minimum comparisons. The proposed technique is named $q-$ *bucket partitioning*, and for each string, it detects all bucket labels with length q and assigns the string to all corresponding buckets. Using bucket label pruning, the algorithm minimizes the number of buckets per string to $\binom{\tau+q}{q}$. To improve verification, the authors of [132] propose the local upper bound pruning method. Additionally, they provide a parallel version of ***Landmark-Join*** using MapReduce, where the data is partitioned using the first Map and the Reducer task.

Another partition-based approach is ***FS-Join*** [167], which focuses on token-based SSJs. ***FS-Join*** introduces a novel partition method, named *vertical partitioning*, which divides each string based on a special set of tokens, called pivots. The segments that belong to the same partition are called fragments. The selection of pivots is decided randomly, by using an even interval technique, or by an even token frequency. ***FS-Join*** uses the prefix filtering for segment intersections in each fragment. Also, the algorithm utilizes a set of filters to further minimize comparisons: (1) string length filtering, (2) segment length filtering, (3) segment intersection filtering and (4) segment difference filtering. In addition to *vertical partitioning*, ***FS-Join*** implements *horizontal partitioning* based on the observation that matched strings have similar lengths.

Neighbor-based approaches The neighborhood-based approaches produce signatures by associating each string with a set of neighbors. As a result, two strings are considered a match if their neighbors have an overlap. For each neighbor-based SSJ, we study the following characteristics:

- the string similarity(ies) supported,
- the algorithm(s) implemented to produce the neighbors,
- the type(s) of filtering used,
- the optimization(s) performed at the verification stage (if any), and
- the parameters used in configuration.

The ***PG-Join*** [95] approach introduces a method of generating signatures for strings based on Graph Proximity Cleansing (GPC) [124]. The authors of [95] present the ***PG-Join*** algorithm that computes signatures using the notion of τ *-neighborhood*: given a threshold τ, the τ*-neighborhood* of a string r is determined as the set of strings whose cardinality of the

Table 3.7: Characteristics of partition-based String Similarity Joins. The ♣ symbol indicates that the corresponding approach implements a parallel version of its algorithm.

Approach	**String Similarities**	**Signatures**	**Substring Selection**	**Filtering Techniques**	**Verification Optimization**	**Parameters**	**Year**
Pass-Join [117]	Levenshtein distance	length and alphabetical order, $\tau+1$ disjoint segments	length-based or position-aware or shift-based multi-match-aware	-	length-aware or extension-based verification	τ	2011
SegFilter [116]	normalized Levenshtein distance	length and alphabetical order, $\tau+1$ disjoint segments	length-based or position-aware or shift-based or multi-match-aware	-	length-aware or extension-based iterative-based or verification	τ	2011
Para-Join♣ [90]	Levenshtein distance	length and alphabetical order, $\tau+1$ disjoint segments	multi-match-aware	content filtering and effective indexing strategy	length-aware or extension-based verification	τ	2013
Pass-JoinK [119]	Levenshtein distance	length and alphabetical order, $\tau+K$ disjoint segments	multi-match-aware	-	length-aware or extension-based verification	τ, K	2014

Table 3.8: Characteristics of partition-based String Similarity Joins. The † symbol indicates that the method is available only as a parallel processing algorithm. The "Filtering Techniques" column has been removed because none of the following partition-based SSJs support filtering.

Approach	**String Similarities**	**Signatures**	**Substring Selection**	**Verification Optimization**	**Parameters**	**Year**
Pass-JoinKMR† [119]	Levenshtein distance	length and alphabetical order, $\tau+K$ disjoint segments	multi-match-aware	extension-based verification	τ, K	2014
Pass-JoinKMRS† [119]	Levenshtein distance	length and alphabetical order, $\tau+K$ disjoint segments	multi-match-aware	length-aware verification	τ, K	2014
ParaLL-Join†[223]	Levenshtein distance	length and alphabetical order, $\tau+1$ disjoint segments and split data into partitions	length-based and position-aware	extension-based verification	τ, number of threads	2017
Pada-Join†[223]	Levenshtein distance	length and alphabetical order, $\tau+1$ disjoint segments and split data into partitions and Spark	length-based and position-aware	extension-based verification and Spark	τ, number of threads	2017

Table 3.9: Characteristics of partition-based String Similarity Joins. The "Parameters " column has been removed since τ is the only configuration parameter of all the following partition-based SSJs.

Approach	**String Similarities**	**Signatures**	**Substring Selection**	**Filtering Techniques**	**Verification Optimization**	**Year**
Bi-Filtering JOIN [81]	Levenshtein distance	length and alphabetical order, $\tau + 1$ segments	re-designed multi-match-aware	forward, backward filtering	extension-based verification	2015
TISM [94]	Levenshtein distance	clustering and blocking of G dataset, use of *exemplars*, $\tau + 1$ segments	even partition	length, count filtering	dynamic programming	2018
SFM [94]	Levenshtein distance	clustering of G dataset, use of *exemplars*, $\tau + 1$ segments	even partition	length, count filtering	dynamic programming	2018
SM [94]	Levenshtein distance	clustering of G dataset, use of *exemplars*, $\tau + 1$ segments	even partition	length, count filtering and character's frequency filtering	dynamic programming	2018

intersection with r is equal to or greater than τ. They implement the ***PG-I*** technique for computing the proximity graph that maps the similarity thresholds τ to the size of the corresponding neighborhood. As a result, the strings are grouped together in GPC clusters and the verification happens exclusively within each cluster, using a unique threshold suitable for the strings it includes. Furthermore, ***PG-Join*** introduces optimization techniques that overcome the issue of constantly re-computing the center of the cluster by incrementally adding new strings to a cluster.

Our next neighbor-based approaches utilize the τ-*variant family* paradigm to generate the signatures of a string. Based on [39], given a string r and a Levenshtein distance threshold τ, the τ-*variant family* includes all deletion neighbors of r by deleting no more than τ characters. The authors of [39] implement a trie structure to store all signatures together (Single Deletion Neighborhoods Trie - ***S-DNT***). The algorithm removes duplicate paths in the trie, by using a compress index, which allows the merging of common subtrees that are identical or isomorphic. Furthermore, they improved ***S-DNT*** by using an incremental similarity join algorithm called ***ISJ-DNT***. To minimize the space requirement, they adopt the non-overlapping length-based partition strategy.

Tree-based approaches

In contrast to the filter-verify approaches, the tree-based approaches do not follow a common implementation prototype. Each tree-based approach uses a tree structure to index the data and perform string comparisons, along with a set of pruning techniques to reduce the number of pairs. Most tree-based approaches do not differentiate between pruning (filtering) and comparison (verification), but perform both simultaneously. For each SSJ that follows the tree-based approach, we study the following characteristics:

- the string similarity(ies) supported,

Table 3.10: Characteristics of partition-based String Similarity Joins. The ♣ symbol indicates that the corresponding approach implements a parallel version of its algorithm. The ♢ symbol refers to the additional characteristics of the parallel version. The † symbol indicates that the method is available only as a parallel processing algorithm.

Approach	**String Similarities**	**Signatures**	**Substring Selection**	**Filtering Techniques**	**Verification Optimization**	**Parameters**	**Year**
MassJoin† [46]	jaccard cosine dice Levenshtein distance	length and alphabetical order, $U+1$ disjoint segments	source: tight upper bound, target: position-aware and multi-match-aware	length and positional filtering and merge-based method and light-weight filter unit	one-time scan of sets, duplicates removal	λ or τ, number of nodes	2014
MLS-Join† [199]	Levenshtein distance	$\tau+1$ disjoint segments, character frequencies, character vectors of length n	multi-match-aware	length and positional filtering	one-time scan of sets, duplicates removal and length-aware verification	λ or τ	2014
Landmark-Join♣ [132]	Levenshtein distance	q disjoint segments	$q-buckets$ partitioning with bucket label pruning	prefix filtering	local upper bound pruning	τ, q number of nodes♢	2014
FS-Join† [167]	dice jaccard cosine	same as [209]	vertical and horizontal partitioning	prefix and string length, segment length, segment intersection, segment difference filtering	-	λ	2017

Table 3.11: Characteristics of neighbor-based String Similarity Joins. The "Verification" column was omitted because none of the following neighbor-based SSJs performed any optimization(s).

Approach	**String Similarities**	**Neighbors**	**Filtering**	**Parameters**	**Year**
PG-Join [95]	normalized $q-gram$ distance	τ -neighborhood: PGC clusters with incremental update of centers and insertion of new strings	comparison of strings within PGC cluster	τ, q	2011
ISJ-DNT [39]	Levenshtein distance	τ-*variant family*: all deletion neighbors of r by deleting no more than τ characters	non-overlapping length based partition	τ	2014

- the tree structure implemented,
- the type(s) of pruning used, and
- the parameters used in configuration.

We begin with **B^{ed}-tree** [226], which provides an indexing structure based on B^{+}-tree, [21] for both Levenshtein distance and normalized Levenshtein distance. Based on the indexing scheme of the B^{+}-tree, the authors of [226] introduce three different mapping functions (or string orders) that project the string domain to the integer domain: dictionary order, gram counting order, and gram location order. All three mapping functions follow a set of properties (comparability, pairwise lower bounding and length bounding), to ensure the efficiency of the string similarity join. Regarding the comparison phase, their verification algorithm tests only the entries on the diagonal of the dynamic programming table, with offset no larger than τ. This idea is based on the insight that any matching of letters with position offset larger than τ leads to a distance of at least $\tau + 1$.

The next tree-based approach we study is ***PeARL*** [161], which uses compressed tries [22] as its indexing structure, focusing on Hamming and Levenshtein distance. The algorithm begins by sorting both input datasets individually in lexicographical order, and partitions them based on shared prefixes. In each data partition, the strings are inserted into an empty trie using preorder DFS traversal. Then, during the matching phase, both tries are traversed concurrently and in cases of unseen nodes, the algorithm proceeds to computing the distance between the nodes. To improve the effectiveness of their algorithms, they authors of ***PeARL*** utilize a set of pruning algorithms: prefix and edit-distance pruning [184], character frequency pruning [4], and $q-gram$ filtering [75]. These pruning algorithms are applied before the verification stage. Additionally, ***PeARL*** uses the τ-banded alignment algorithm [61], which computes the distances between two strings more efficiently. Additionally, the authors of ***PeARL*** offer a parallelization with an in-memory MapReduce [45] version of their algorithm that simultaneously processes each data partition by assigning it to an independent mapper.

Another family of tree-based approaches that utilize a trie index is ***Trie-Join*** [59]. The authors of [59] propose a novel approach based on the assumption that iff a string is not considered a match to a node, then it will not be matched with the strings under that node. Thus, further search and comparison can be avoided. ***Trie-Join*** consists of a set of algorithms: (1) Trie-Search, which utilizes sub-trie pruning, (2) Trie-Traverse, utilizing dual sub-trie pruning to avoid the duplicate computation of Trie-Search, (3) Trie-Dynamic that avoids the redundant active node computation introduced in Trie-Traverse, (4) Trie-PathStack, which minimizes the memory consumption introduced in Trie-Dynamic, and (5) Bi-Trie-PathStack, an extension of Trie-PathStack that handles large Levenshtein distance thresholds. These algorithms were proposed as part of solving the self-join problem ($R = Q$), but can be easily extended to support $R \neq Q$. To further improve the performance of their algorithms, the authors introduced three pruning techniques to reduce the size of the active nodes: (1) length pruning, (2) single-branch pruning, and (3) count pruning. Additionally, the ***Trie-Join*** family of algorithms provides support for dynamic updates in the trie.

Our final tree-based method, ***PreJoin*** [71], introduces the idea of combining dynamic preorder traversal [59] with an active node generation method. The authors of [71] avoid latter pruning by forcing heuristics early on the algorithm to minimize the size of the generated active nodes. In contrast to Trie-Traverse, ***PreJoin*** uses a non-fixed traversal order that prioritizes *significant sub-tries* over sub-tries imposed by the trie order. For the dynamic re-ordering of sub-tries, ***PreJoin*** introduces three different ordering methods: (1) based on the size of the sub-trie rooted at each child, (2) based on the fan-out value of each child of a traversing node, and (3) based on the depth of the sub-trie rooted at each child of a traversing node. Additionally, the authors of [71] propose an extension of the main algorithm, named ***PreJoinPlus***, which partitions the search string space to speed up the computations for large Levenshtein distance

[21]`https://en.wikipedia.org/wiki/B%2B_tree`
[22]`https://en.wikipedia.org/wiki/Trie`

thresholds based on [30].

Table 3.12: Characteristics of tree-based String Similarity Joins. The ♣ symbol indicates that the corresponding approach implements a parallel version of its algorithm. The ♢ symbol refers to the additional characteristics of the parallel version.

Approach	**String Similarities**	**Tree Type**	**Pruning**	**Parameters**	**Year**
B^{ed}-tree [226]	Levenshtein distance, normalized Levenshtein distance	B^{+}-tree	-	τ	2010
PeARL♣ [161]	Hamming distance Levenshtein distance	Trie	prefix, edit-distance and character frequency pruning, $q - gram$ filtering	τ, number of threads♢	2011
Trie-Join [59]	Levenshtein distance	Trie	length pruning, single-branch pruning, count pruning	τ	2012
PreJoin [71]	Levenshtein distance	Trie	early rules for active node generation, dynamic pre-ordering of sub-tries, investigation of relatively deeper sub-tries	τ	2017
PreJoinPlus [71]	Levenshtein distance	Trie	early rules for active node generation, dynamic pre-ordering of sub-tries, investigation of relatively deeper sub-tries and partition of string search space based on [30]	τ	2017

3.1.6 Evaluation of String Similarity Joins for Link Discovery

After presenting the results of our systematic survey for SSJs for LD in Section 3.1.5, we now address the research question $CS - Q_3$ posited in Section 3.1.3, by discussing and analyzing the evaluation results found in the corresponding papers. Note that we are interested in the comparison of the different SSJs approaches, and thus, we do not include results of (1) string similarity searches, (2) hyper-parameter optimizations, (3) comparisons between the different filtering/pruning/index/verification strategies of the same method. In case of the latter, we do mention the strategies used for the method to be compared with other SSJs when available.

To answer the research question $CS - Q_3$ presented in Section 3.1.3, we divide the SSJs introduced in Section 3.1.5 based on the category of string similarities they target. We report experimental results of the runtime of SSJs for a particular string similarity and threshold, unless specified otherwise in parenthesis. For each SSJ J_1, the runtime results were either obtained from the corresponding paper of J_1, or from the publication of an SSJ J_2, where the SSJ J_2 was compared against J_1 (Tables 3.14- 3.35). Incase of the latter, we include an extra column with the *vs.* tag to show the name of the SSJ J_2 and indicate the publication from which the values were extracted. We include the specific values when they are available (number in italics), otherwise we approximate the values based on the corresponding plots. The authors of some SSJs have presented results for only a subset of the string similarities that their approach improves. Published results that do not include runtime experiments for any string similarity are not presented. Note that the experiments were carried out using datasets that vary in nature and size, different string similarities and experimental set-ups. All SSJ experiments were conducted for the self-join case, unless otherwise specified. Finally, we include *only* runtime results of SSJs presented throughout Section 3.1.5.

Results for Token-based String Similarities To avoid repetition, we present the datasets that were used throughout the evaluation of the token-based SSJs in Table 3.13 and provide a short description of their characteristics. For a dataset to be included in our list, it needs at least 2 SSJs from Section 3.1.5 compared against it.

We collected and present the experimental results for numerous SSJs for a set of token-based string similarities in Tables 3.14- 3.24. Our first general observation is that ***PPJoin*** and ***PPJoin+*** are considered state of the art when it comes to token-based string similarity joins. Introducing positional and suffix filtering and combining it with prefix filtering has been a novel idea that served as inspiration to many newer SSJs (e.g. [162, 214]), and has been used extensively in LD frameworks [134]. Based on [221] and Tables 3.14, 3.15, 3.18 and 3.20, ***PPJoin+*** outperforms ***PPJoin*** in most experimental set-ups, since the suffix filtering prunes candidate pairs further, and thus minimizes the verification step runtime.

Beginning with the jaccard and cosine similarity, we present the results of 3 approaches that were tested against ***PPJoin*** and ***PPJoin+***: (1) ***MPJoin***, (2) ***MGJoin*** and (3) ***Adapt-Join***. As we already mentioned in Section 3.1.5, all these SSJs build upon and extend the main idea of ***PPJoin*** and ***PPJoin+***, and have successfully outperformed their predecessors in terms of effectiveness (Tables 3.14, 3.15, 3.18). Regarding the scalability of the methods, we notice that in all datasets, ranging from 10,000 to 7,844,465 records, all 3 methods scale better than ***PPJoin*** and ***PPJoin+***. ***MPJoin*** introduced the min-prefix filtering technique as a generalization of prefix filtering, which keeps the size of the inverted indexes at a minimum at all times; ***MGJoin*** implements multi-prefix filtering with multiple global orderings, and ***Adapt-Join*** utilizes an adaptive selection of prefixes for strings based on the estimation of the candidate size.

The second group of approaches that have optimized jaccard similarity includes ***MassJoin***, ***FS-Join***, ***V-Smart-Join*** and ***EFS-S***, all implemented using MapReduce. Based on the discussion in [167], ***FS-Join*** was created to address the following issues of the other 3 approaches: (1) similarity computation of the same strings that appear more than once, and (2) the load balance problem, where Reduce tasks with the same key have different sizes (discussed earlier in this chapter). Based on the results in Tables 3.16 and 3.17, we observe that ***FS-Join*** outperforms the other 3 approaches, with ***V-Smart-Join*** having the worst runtime performance. Additionally, ***V-Smart-Join*** does not incorporate a filtering technique due to the memory overhead created by storing large indexes in memory. Additionally, both ***MassJoin*** and ***V-Smart-Join*** run out of memory for the experiments of the larger datasets - ENRON$_{0.5M}$, PubMed(Abstract)$_{7.4M}$ and Wiki(Abstract)$_{4.3M}$. Note that ***V-Smart-Join*** outperforms ***VCL (PPJoin+)*** for the jaccard similarity co-efficient (Table 3.22), however the authors of [127] conducted experiments utilizing 500 Machines with 1GB memory and 10GB disk space each, (due to the nature and the size of the dataset), while in [167], the experiments were conducted using 11 nodes with 15GB of memory and 4 virtual cores.

SSJ-2 and ***SSJ-2R*** have also conducted a set of experiments using cosine similarity. Both methods are based on ***EFS-S*** and ***PairWiseMR***, using the signature scheme of the first and the verification optimization of the second. Based on Table 3.19, both approaches have outperformed their baselines with ***SSJ-2*** and ***SSJ-2R*** being 2 and 4.5 times faster resp. than ***EFS-S***. Based on the results in [17] the exploitation of the remainder file proved to be a significant improvement, thus ***SSJ-2R*** is always more time-efficient than ***SSJ-2***.

Table 3.13: Evaluation Dataset Characteristics for Token-based String Similarity Joins

Name	Type	Size	Source
DBLP(a+t)$_{0.1M}$	bibliography records, concatenation of author name(s) and the title of a publication	100,000	[162]
DBLP(a+t)$_{0.9M}$	bibliography records, concatenation of author name(s) and the title of a publication	900,000	[221]
DBLP(a+t)-5GRAM	bibliography records, concatenation of author name(s) and the title of a publication	900,000	[221]
DBLP(a+t)$_{1.1M}$	bibliography records, concatenation of author name(s) and the title of a publication	1,021,062	[168]
5xDBLP	bibliography records, concatenation of author name(s) and the title of a publication	5,105,310	[168]
DBLP(author)	bibliography records, name(s) of author	613,542	[213]
DBLP(title)	bibliography records, title of a publication	10,000	[95]
ENRON$_{0.5M}$	Enron email collection	500,000	[167]
ENRON$_{10K}$	Enron email collection	10,000	[167]
IMDB(title+actor))$_{0.1M}$	movie records, concatenation of movie names and the actor(s)	100,000	[162]
IMDB(title+actor))$_{1.5M}$	movie records, concatenation of movie names and the actor(s)	1,568,893	[168]
5xIMDB	movie records, concatenation of movie names and the actor(s)	7,844,465	[168]
Twitter	tweets	2,753,005	[168]
PubMed(Abstract)$_{2.3M}$	abstracts of medical articles	2,347,362	[46]
PubMed(Abstract)$_{7.4M}$	abstracts of medical articles	7,400,308	[167]
PubMed(Abstract)$_{7.4K}$	abstracts of medical articles	7,400	[167]
Wiki(Abstract)$_{4.3M}$	abstracts from Wikipedia dumps	4,305,022	[167]
Wiki(Abstract)$_{4.3K}$	abstracts from Wikipedia dumps	4,300	[167]
TREC WT10-63K	english language documents	63,126	[17]
Oxford Misspellings	natural language collection of unique misspellings	39,030	[95]
Delicious Tags	bookmark tags	48,397	[95]
IPs$_1$	internet cookies	133,000,000	[127]
CiteSeer	bibliography records	568,237	[169]
CONF	conference names	10,000	[160]
USPS	person names, street names, city names, states, and zip codes	-	[160]
AOL Query Log	distinct real keyword queries	1,000,000	[213]

Regarding the neighbor-based SSJ approaches, ***PG-Join*** was tested using its algorithm ***PG-I*** against the state-of-the-art, ***PG-Skip*** [96], using the normalized $q-grams$ distance. Based on Table 3.21, the idea of introducing an adaptive threshold method that is solely influenced by underlying data distribution, and using the results of smaller neighborhoods to incrementally compute larger ones proved to be beneficial. ***PG-I*** is 2.5 times faster than ***PG-Skip*** on the largest dataset, since ***PG-Skip*** needs to compute the approximate proximity graph in advance, and not update it incrementally as does ***PG-I***.

On a similar note, [169] also investigated the idea of using different thresholds for each string by building two different index structures to support their cause. Based on Tables 3.23 and Tables 3.24, the authors of [169] studied 3 different distributions, Normal, Uniform and Poisson, and tested both ***PBIJoin*** and ***SAIJoin*** against ***PPJoin+*** and ***Adapt-Join***. ***SAIJoin*** outperformed all other methods, regardless of the selected distribution and string similarity. This is mainly due to its similarity-aware inverted index that probes a small part of the related lists and generates the best performance. The most important observation from [169] is that ***PPJoin+*** and ***Adapt-Join*** both order the strings based on their lengths, which does not indicate that strings with similar lengths require similar thresholds.

Table 3.14: Runtime results for SSJs using jaccard similarity with threshold 0.8, for the 5 variations of the DBLP(a+t) dataset.

SSJ	vs.	DBLP(a+t)$_{0.1M}$	DBLP(a+t)$_{0.9M}$	DBLP(a+t)-5GRAM	DBLP(a+t)$_{1.1M}$	5xDBLP
PPJoin		-	5	33	-	-
	MGJoin	-	-	-	18	180
PPJoin+		-	4	28	-	-
	MPJoin	500	-	-	-	-
	MGJoin	-	-	-	10	100
MPJoin$(q=2)$		200	-	-	-	-
MGJoin		-	-	-	7	60

Table 3.15: Runtime results for SSJs using jaccard similarity with threshold 0.8, for the 3 variations of the IMDB(a+t) dataset, the Twitter dataset and the ENRON$_{0.5M}$ dataset.

SSJ	vs.	IMDB(a+t)$_{0.1M}$	IMDB(a+t)$_{1.5M}$	5xIMDB	Twitter	ENRON$_{0.5M}$
PPJoin		-	-	-	-	34
	MGJoin	-	35	400	50	-
	Adapt-Join	-	-	-	-	200
PPJoin+		-	-	-	-	29
	MPJoin	150	-	-	-	-
	MGJoin	-	32	430	40	-
	Adapt-Join	-	-	-	-	200
MPJoin$(q=2)$		100	-	-	-	-
MGJoin		-	28	220	25	-
Adapt-Join		-	-	-	-	100

Results for Character-based String Similarities To avoid repetition, we present the datasets that were used throughout the evaluation of the token-based SSJs in Table 3.13 and provide a short description of their characteristics. For a dataset to be included in our list, there needed to be at least 2 SSJs from Section 3.1.5 compared against it.

Table 3.16: Runtime results for SSJs using jaccard similarity with threshold 0.8, for the 3 variations of the PubMed(Abstract) dataset. ***V-Smart-Join*** uses the Online Aggregation (OA) method and ***MassJoin*** uses the Merge+Light version of the approach. The ✧ symbols indicate that the SSJ could not complete the experiment due to running out of memory. The ✳ symbols indicate that the SSJ could not run successfully or completely on the corresponding datasets.

SSJ	vs.	PubMed(Abstract)$_{7.4K}$	PubMed(Abstract)$_{2.3M}$	PubMed(Abstract)$_{7.4M}$
MassJoin (M+L)		-	800	-
	FS-Join	250	-	✳
FS-Join		200	-	450
V-Smart-Join (OA)	***FS-Join***	✳	-	✳
	MassJoin	-	✧	-
EFS-S	***FS-Join***	200	-	700
	MassJoin	-	2,000	-

Table 3.17: Runtime results for SSJs using jaccard similarity with threshold 0.8, for the 2 variations of the ENRON dataset and 2 variations of the Wiki(Abstract) dataset. ***V-Smart-Join*** uses the Online Aggregation (OA) method and ***MassJoin*** uses the Merge+Light version of the approach. The ✧ symbols indicates that the SSJ could not complete the experiment due to running out of memory. The ✳ symbols indicates that the SSJ could not run successfully or completely on the corresponding datasets.

SSJ	vs.	ENRON$_{10K}$	ENRON$_{0.5M}$	Wiki(Abstract)$_{4.3K}$	Wiki(Abstract)$_{4.3M}$
MassJoin (M+L)		-	600	-	-
	FS-Join	300	✳	400	✳
FS-Join		200	500	100	200
V-Smart-Join (OA)	***FS-Join***	500	✳	1,600	✳
	MassJoin	-	✧	-	-
EFS-S	***FS-Join***	250	2,600	100	450
	MassJoin	-	7,000	-	-

Table 3.18: Runtime results for SSJs using cosine similarity with threshold 0.8 for the 3 variations of the DBLP(a+t) dataset and the ENRON$_{0.5M}$ dataset.

SSJ	vs.	DBLP(a+t)$_{0.9M}$	DBLP(a+t)-5GRAM	DBLP(a+t)$_{1.1M}$	ENRON$_{0.5M}$
		13	100	-	120
PPJoin	***Adapt-Join***	-	-	100	-
	MGJoin	-	-	110	-
		8	90	-	110
PPJoin+	***Adapt-Join***	-	-	100	-
	MGJoin	-	-	60	-
Adapt-Join		-	-	100	-
MGJoin		-	-	40	-

Similarly to the token-based string similarities, we have collected and present experimental results for SSJs that target character-based string similarities in Tables 3.26- 3.33. Our first observation is that all SSJs have focused on optimizing the Levenshtein distance, and only ***PSH*** and ***REEDED*** have been tested against Damerau-Levenshtein and weighted Levenshtein

Table 3.19: Runtime results for SSJs using cosine similarity with threshold 0.8, unless specified otherwise. For each SSJ, the results were obtained from either its corresponding original publication or from the corresponding original publication of another SSJ that the initial SSJ was compared against (*vs.* tag).

SSJ	vs.	TREC WT10-63K
PairWiseMR ($\lambda = 0.9$)	***SSJ-2,SSJ-2R***	*51,540*
EFS-S ($\lambda = 0.9$)	***SSJ-2,SSJ-2R***	*28,534*
SSJ-2 ($\lambda = 0.9$)		*20,994*
SSJ-2R ($\lambda = 0.9, K = 4$)		*9,745*

Table 3.20: Runtime results for ***PPJoin*** and ***PPJoin+*** using the weighted cosine similarity with threshold 0.8. The results for all SSJs were obtained from [221].

SSJ	DBLP(a+t)$_{0.9M}$	DBLP(a+t)-5GRAM	ENRON$_{0.5M}$
PPJoin	6.1	98.0	40.1
PPJoin+	6.0	95.0	40.0

Table 3.21: Runtime results in seconds for ***PG-Join*** with ***PG-I*** and ***PG-Skip*** [96], using the normalized $q - grams$ distance with adaptive threshold. For the Oxford Misspellings and Delicious Tags, the parameter q was set to 2, and for DBLP (title), it was set to 3. The results for all SSJs were obtained from [95].

SSJ	Oxford Misspellings	Delicious Tags	DBLP (title)
PG-I	115	350	280
PG-Skip	165	650	750

Table 3.22: Runtime results for ***V-Smart-Join*** and ***VCL*** using Ruzicka similarity (Jaccard similarity coefficient) with threshold 0.8, utilizing 500 machines. The results for all SSJs were obtained from [127].

SSJ	IPs$_1$
V-Smart-Join with Online-Aggregation	800
V-Smart-Join with Lookup	1,000
V-Smart-Join with Sharding	1,300
VCL (PPJoin+)	4,000

distance (Table 3.33). In both publications, the proposed approaches outperformed the corresponding state of the art, showing the importance of implementing more SSJs suitable for other edit-distance similarities.

Throughout our systematic survey, we notice a set of SSJs whose presence in the experimental evaluation was more prominent, as many authors of different publications chose to compare their approaches against them. ***Ed-Join***, the most popular, has served as the state of the art over the last decade in optimization of the Levenshtein distance. The novel idea of location-based

Table 3.23: Runtime results for SSJs using various jaccard similarity threshold distributions. The results for all SSJs were obtained from [169].

SSJ	CiteSeer ⋈ DBLP(a+t)$_{1.1M}$			CiteSeer			DBLP(a+t)$_{1.1M}$		
	Normal	Uniform	Poisson	Normal	Uniform	Poisson	Normal	Uniform	Poisson
PPJoin+	15	27	59	54	65	85	12	17	32
Adapt-Join	14	24	51	50	60	80	8	15	28
PBIJoin	12	22	42	38	49	68	7	14	26
SAIJoin	11	18	36	35	40	57	6	12	22

Table 3.24: Runtime results for SSJs using various cosine similarity threshold distributions. The results for all SSJs were obtained from [169].

SSJ	CiteSeer ⋈ DBLP(a+t)$_{1.1M}$			CiteSeer			DBLP(a+t)$_{1.1M}$		
	Normal	Uniform	Poisson	Normal	Uniform	Poisson	Normal	Uniform	Poisson
PPJoin+	60	120	290	275	300	370	35	60	122
PBIJoin	50	90	200	235	275	340	30	52	108
SAIJoin	25	50	110	100	140	225	22	35	72

mismatch filtering combined with content-based mismatch filtering proved to be a powerful technique that minimized the set of candidate pairs and sped up the verification process. Over the years, it has been compared against many filter-verify and tree-based SSJ approaches such as ***VChunkJoin***, ***PassJoin*** and **B^{ed}-tree**. Based on Tables 3.26, 3.27, 3.31 and 3.32, we notice that in datasets with short strings, ***Ed-Join*** uses small values for q (limited pruning power). In combination with large thresholds, it creates a larger number of candidate pairs that negatively affect the verification and overall runtime [213, 117]. In cases of datasets with long strings, where q is set to values ≥ 4, using large Levenshtein distance thresholds results in ***Ed-Join*** outperforming both ***Trie-Join*** and **B^{ed}-tree** [59, 226].

The second SSJs we investigate are ***Trie-Join*** and **B^{ed}-tree** (Tables 3.26, 3.27, 3.31 and 3.32). The experimental findings of [81, 215, 226] support the existing evidence that tree-based SSJs are not suitable for datasets with short strings that share a small number of common prefixes, since it is expensive to traverse the resulting large trie structure. However, in cases of short strings with many shared common prefixes, both tree-based approaches outperformed ***Ed-Join*** and ***Adapt-Join*** in terms of efficiency [214]. To overcome the issue of dealing with short strings, the authors of ***Trie-Join*** showed that for datasets with short strings, where the threshold values must stay low ($\tau \leq 3$), it is more suitable to use the Trie-PathStack, and in case of long strings with larger τ, it is advisable to use the Bi-Trie-PathStack [59]. In addition to the experiments conducted in [59], the authors of ***PreJoin*** extended their experimental set-up for $\tau \geq 4$ and showed that prioritizing significant sub-tries over sub-tries imposed by the trie order leads to more effective pruning and efficient verification [71].

Regarding the partition-based SSJs, we notice that ***Pass-Join*** outperforms both prefix and tree-based approaches, regardless of the lengths of the dataset's strings [117]. For short strings, ***Pass-Join*** utilizes segments to prune large numbers of mismatched pairs and the segments are selected across the strings and not restricted to prefix filtering. For long strings, ***Ed-Join*** needed to use a mismatch technique and ***IndexChunk*** needed to use an error estimation-based filter in the verification phase, which were inefficient while ***Pass-Join***'s verification method was more efficient [117]. As we discussed in Section 3.1.5, most partition-based approaches were

Table 3.25: Evaluation Dataset Characteristics for Character-based String Similarity Joins

Name	Type	Size	Source
DBLP(a+t)$_{0.9M}$	bibliography records, concatenation of author name(s) and the title of a publication	900,000	[117]
DBLP(a+t)$_{1.1M}$	bibliography records, concatenation author name(s) and the title of a publication	1,021,062	[214]
DBLP(a+t)$_{3.2M}$	bibliography records, concatenation author name(s) and the title of a publication	3,203,996	[199]
DBLP(author)	bibliography records, name(s) of author	613,542	[213]
DBLP(title)-small	bibliography records, title of a publication	2,616	[194]
ACM(author)	bibliography records, name(s) of author	2,295	[194]
GoogleProducts	products name(s) from Google	3,226	[194]
ABT	E-commerce products description(s)	1,081	[194]
TEXAS-5GRAM	Broker and Sales licensees database from the Texas Real Estate Commission, concatenation of 19 attributes, including person names, addresses, and licence information	150,000	[219]
TREC-8GRAM	TREC-9 Filtering Track Collections, references from the MEDLINE database	350,000	[219]
UNIREF$_{0.3M}$	protein sequences in flat text format from UNIPROT	377,438	[155]
UNIREF$_{0.5M}$	protein sequences in flat text format from UNIPROT	508,038	[226]
IMDB(actor)$_{1.2M}$	movie records, concatenation of actor(s) names	1,200,000	[215]
IMDB(title)$_{1.5M}$	movie records, concatenation of movie names	1,568,893	[226]
LEXICON	Gene/Protein lexicon generated from MEDLINE documents	473,428	[215]
QueryLog-small	collection of query strings that were randomly chosen from the AOL Query Log	464,189	[117]
AOL Query Log	distinct real keyword queries	1,208,844	[214]
PhilAdd	Philadelphia addresses	547,771	[92]
GBEST	gene sequence obtained from the Expressed Sequence Tags database of NCBI GenBank	1,020,109	[199]
Word	english words	146,033	[81]
ENGLISH-DICT	english words	150,000	[39]

built upon and extended the ***Pass-Join*** framework, including the ***Bi-Filtering JOIN***. Based on the experimental results of [81], even though ***Bi-Filtering JOIN***'s bi-directional filtering created more overhead than forward only filtering, backward filtering reduced the size of the candidate set, and effectively minimized the time required for comparisons. On a similar note, based on [119], ***Pass-JoinK*** outperformed *Pass-Join*, proving that a method that partitions the data even further is beneficial to the efficiency of the algorithm without increasing false negative pairs. For large thresholds, the multi-threading idea presented in [223] showed the necessity of using multiple cores for string similarity joins, since *ParaLL-Join* outperformed ***Pass-Join*** in all experimental settings for $\tau \geq 3$. Finally, any parallel extension of ***Pass-Join***, such as ***Pass-JoinKMR***, ***Pass-JoinKMRS*** and ***Pada-Join*** (Tables 3.28 and 3.29), improved the overall runtime of the corresponding filter-verify framework only for large datasets, and based on [223], should be avoided when the dataset includes less than 1 million strings.

Table 3.26: Runtime results for SSJs using the Levenshtein distance with $\tau = 2$ (unless specified otherwise), for the 3 variations of the DBLP dataset.

SSJ	vs.	DBLP(a+t)$_{0.9M}$	DBLP(a+t)$_{1.1M}$	DBLP(author)
Ed-Join		3.00	-	-
	VChunkJoin	0.85	-	-
	PassJoin	70.00	-	700.00
	SegFilter	1.70	-	2.70
	PassJoinK ($\tau = 4, K = 2$)	50.00	-	-
	TrieJoin	($\tau = 4, q = 6$) 1.95	-	($q = 3$) 2.30
	Adapt-Join	-	300.00	-
	IndexGram	-	1.70	-
	PreJoin	-	-	($q = 3$) 1.48
	Partition-NED	-	-	($\tau = 8, \eta = 0.8$) 3.30
Ed-Join-I		3.00	-	-
All-Pairs-Ed		6.00	-	-
	TrieJoin	($\tau = 4, q = 6$) 1.95	-	($q = 3$) 2.78
Adapt-Join		-	200.00	-
IndexGram		-	1.60	-
IndexChunk		-	0.90	-
	SegFilter	1.70	-	3.00
Partition-NED($\tau = 8, \eta = 0.8$)		-	-	2.30
B^{ed}-tree	***VChunkJoin***	2.78	-	-
	IndexGram	2.95	-	-
Trie-Join		($\tau = 4$) 2.30	-	2.48
	PassJoin	80.00	-	50.00
	SegFilter	1.70	-	1.30
	VChunkJoin	-	1.90	-
	Adapt-Join	-	200.00	-
	IndexGram	-	1.70	-
	PreJoin	-	-	1.00
PreJoin		-	-	0.90
PreJoinPlus		-	-	1.00

Results for Hybrid String Similarities Based on findings in Section 3.1.5, there have been only two publications that focus on improving the runtime of hybrid string similarities - [121] and [213].

Begining with [121], the authors conducted three sets of experiments, measuring the efficiency and effectiveness of (1) the similarity measures, (2) their SSJs and (3) the different signatures' selection methods. They compared their methods with a jaccard implementation, named JaccT [11] rather than other SSJs. For their experiments they used three datasets, and for each dataset they created one source dataset with 100 strings and a target dataset with 200 strings. Based on Table 3.34, they supported their initial hypothesis that using synonyms to expand a string's representation is beneficial for accuracy and generates low overheads in runtime. This is supported by the fact that ***SI-Join***, with full expansion, scales linearly with the size of the join data sizes and number of synonyms. Additionally, it achieves the highest filtering ratio[23] and the lowest candidate size and thus, outperforms the other SSJs in both completeness and efficiency. Furthermore, they conducted a set of experiments incorporating LSH filtering into the baseline (Table 3.34). Based on their results, LSH filtering is beneficial

[23]number of pruned string pairs divided by the total number of string pairs

Table 3.27: Runtime results for SSJs using the Levenshtein distance with $\tau = 2$ (unless specified otherwise), for the 2 variations of the AOL Query Log dataset.

SSJ	vs.	AOL Query Log	QueryLog-small
Ed-Join		-	-
	VChunkJoin	-	-
	PassJoin	-	15.00
	SegFilter	-	1.18
	PassJoinK ($\tau = 4, K = 2$)	120.00	-
	TrieJoin	($\tau = 3, q = 2$) 3.30	-
	Adapt-Join	500.00	-
	IndexGram	-	-
	PreJoin	($q = 3$) 1.00	-
	Partition-NED	($\tau = 8, \eta = 0.8$) 3.85	-
All-Pairs-Ed	***TrieJoin***	($\tau = 3, q = 2$) 3.85	-
Adapt-Join		500.00	-
IndexChunk	***SegFilter***	-	1.18
Partition-NED($\tau = 8, \eta = 0.8$)		2.70	-
Trie-Join		($\tau = 3$) 2.95	-
	PassJoin	-	20.00
	SegFilter	-	1.18
	VChunkJoin	-	-
	Adapt-Join	400.00	-
	PreJoin	1.34	-
PreJoin		1.00	-
PreJoinPlus		1.30	-

Table 3.28: Runtime results for SSJs using the Levenshtein distance with $\tau = 2$ (unless specified otherwise), for the 4 variations of the DBLP dataset.

SSJ	vs.	DBLP(a+t)$_{0.9M}$	DBLP(a+t)$_{1.1M}$	DBLP(author)
Pass-Join		5.00	-	20.00
	ParaLL-Join	50.00	-	-
	Pass-JoinK($K = 2, \tau = 4$)	10.00	-	-
	Bi-Filtering JOIN	-	20.00	-
SegFilter		0.70	-	1.30
Pass-JoinK($K = 2, \tau = 4$)		10.00	-	-
Pass-JoinKMR($K = 1, \tau = 4$)		500.00	-	-
Pass-JoinKMRS($K = 1, \tau = 4$)		250.00	-	-
ParaLL-Join		27.00	-	-
Pada-Join		-	-	-
Bi-Filtering JOIN		-	8.00	-

only for thresholds above 0.8; therefore, LSH filtering has a greater pruning power than prefix filtering, but comes at the cost of filtering time.

Continuing with the second publication, the authors of [213] conducted a set of experiments using two datasets - DBLP(author) with size of 600,000 and AOL Query Log with size of 1 million. For their fuzzy token-based string similarities, they compared ***Fast-Join***, with token-sensitive filtering technique, against prefix-filtering for different fuzzy-jaccard similarity

Table 3.29: Runtime results for SSJs using the Levenshtein distance with $\tau = 2$ (unless specified otherwise), for the 2 variations of the AOL Query Log dataset.

SSJ	vs.	AOL Query Log	QueryLog-small
Pass-Join		-	2.00
	ParaLL-Join	-	23.00
	Pass-JoinK$(K = 2, \tau = 4)$	100.00	-
	Bi-Filtering JOIN	-	-
SegFilter		-	0.60
Pass-JoinK$(K = 2, \tau = 4)$		80.00	-
Pass-JoinKMR$(K = 1, \tau = 4)$		400.00	-
Pass-JoinKMRS$(K = 1, \tau = 4)$		250.00	-
ParaLL-Join		-	14.00
Pada-Join		-	100.00

Table 3.30: Runtime results for ***MassJoin*** and ***MLS-Join*** using the Levenshtein distance with $\tau = 2$, for the DBLP(a+t)$_{3.2M}$ dataset and the GBEST dataset as published in [199].

SSJ	DBLP(a+t)$_{3.2M}$	GBEST $(\tau = 4)$
MassJoin	0.8×10^7	0.7×10^7
MLS-Join	0.2×10^7	0.2×10^7

Table 3.31: Runtime results for SSJs using the Levenshtein distance with $\tau = 2$ (unless specified otherwise), for the TEXAS-5GRAM dataset and the TREC-8GRAM dataset.

SSJ	vs.	TEXAS-5GRAM	TREC-8GRAM	UNIREF$_{0.3M}$	UNIREF$_{0.5M}$
Ed-Join		0.30	1.00	-	-
	VChunkJoin	-	$(\tau = 4)$ 0.50	-	$(\tau = 4)$ 0.80
	IndexGram	-	-	$(\tau = 4)$ 2.26	-
	B^{ed}-tree	-	-	-	800.00
Ed-Join-I		0.3	1	-	-
All-Pairs-Ed		0.8	1	-	-
VChunkJoin		-	$(\tau = 4)$ 0.10	-	$(\tau = 4)$ 0.10
IndexGram		-	-	$(\tau = 4)$ 2.00	-
IndexChunk		-	-	$(\tau = 4)$ 1.85	-
B^{ed}-tree		-	-	-	120.00
	IndexGram	-	-	$(\tau = 4)$ 2.20	-

thresholds. Based on Table 3.35, the token-sensitive filtering generates a smaller amount of signatures and candidate pairs compared to prefix filtering, and is 3 to 5 times faster. Based on their experimental results, ***Fast-Join*** requires constant time to generate signatures and compute the set of candidate pairs, whereas the verification time is influenced by the different threshold values. However, for the largest dataset (AOL Query Log), the verification time decreases significantly as the threshold values increase.

Table 3.32: Runtime results for SSJs using the Levenshtein distance with $\tau = 2$ (unless specified otherwise), for the 2 variations of the IMDB dataset, the LEXICON dataset, the Word dataset and the ENGLISH-DICT dataset.

SSJ	vs.	IMDB(actor)$_{1.2M}$	IMDB(title)$_{1.5M}$	LEXICON	Word	ENGLISH-DICT
Ed-Join	***VChunkJoin***	2.90	-	-	-	-
	B^{ed}-tree	-	200.00	-	-	
	Trie-Join	-	-	-	-	($\tau = 1, q = 3$) 1.00
All-Pairs-Ed	***Trie-Join***	-	-	-	-	($\tau = 1, q = 3$) 1.00
VChunkJoin		1.85	-	1.04	-	-
B^{ed}-tree		-	2,000.00	-	-	-
	VChunkJoin	-	-	2.95	-	-
Trie-Join		-	-	-	-	0.80
	VChunkJoin	2.30	-	1.70	-	-
	ISJ-DNT	-	-	-	-	1.30

Table 3.33: Runtime results for ***PassJoin*** and ***REEDED*** using the weighted Levenshtein distance with $\tau = 2$, for the DBLP(title)-small, ACM(author), GoogleProducts and ABT datasets as published in [194]. Runtime results for ***PSH*** and ***Trie-Join*** using Damerau-Levenshtein edit distance with $\tau = 2$, for the PhilAdd dataset as published in [92].

SSJ	DBLP(title)-small	ACM(author)	GoogleProducts	ABT	SSJ	PhilAdd
REEDED	*15.27*	*8.54*	*20.43*	*27.71*	***PSH***	*216,889*
PassJoin	*30.74*	*18.53*	*62.31*	*140.73*	***Trie-Join***	*82,660*

Table 3.34: Runtime results for JaccT, ***SN-Join*** and ***SI-Join*** for the CONF and USPS datasets as published in [121]. All experiments were carried out using the jaccard similarity with threshold 0.8. JaccT-L, ***SN-Join***-L and ***SI-Join***-L incorporate the LSH filtering technique. The LSH filtering parameters for the CONF dataset are k=3, l=3, and for the USPS dataset are k=4, l=3. The runtimes are reported in seconds, for the maximum join data size, 10 and 1,000 for the CONF and USPS datasets resp.

Method	CONF	USPS	Method	CONF	USPS
JaccT	27	450	JaccT-L	15	320
SN-Join with FE	18	350	***SN-Join***-L with FE	12	250
SN-Join with SE	22	370	***SN-Join***-L with SE	13	270
SI-Join with FE	**3**	**40**	***SI-Join***-L with FE	**2**	**40**
SI-Join with SE	6	70	***SI-Join***-L with SE	4	60

Based on our analysis for SSJs for hybrid similarities, as well as Tables 3.34 and 3.35, we make two important observations: (1) introducing synonyms into existing string similarities is proven to be beneficial without sacrificing the effectiveness of the SSJ; and (2) building upon and extending the existing prefix filtering technique leads to improvements regarding the pruning power of an SSJ while maintaining low overheads. Even though the results of [121] and [213] are encouraging and have shown that hybrid string similarities are able to successfully capture both syntactic and semantic variations of strings, they have been little investigated.

Table 3.35: Runtime results for ***Fast-Join*** (token-sensitive) against prefix filtering for the DBLP(author) and AOL Query Log datasets as published in [213]. All experiments were carried out using the fuzzy jaccard similarity with threshold 0.8. The runtimes are reported in seconds and η was set to 0.8.

	DBLP(author)	AOL Query Log
Fast-Join (token-sensitive)	**30**	**100**
prefix filtering	135	550

3.2 Time Relations for Link Discovery

The first LD framework that provided temporal LD for RDF datasets was Silk, and it did so by incorporating the recently published work of Smeros et al. [191]. The authors of this paper used *MultiBlock* to develop an approach for the efficient computation of temporal links. As shown in Section 4.4, Aegle was able to outperform this approach by 4 orders of magnitude.

In the field of stream reasoning and Complex Event Processing (CEP) on Linked Data, there has been a notable amount of research into querying temporal data. For example, *Continuous SPARQL* (C-Sparql) [18] provides a syntactic and semantic extension of SPARQL to query RDF temporal data by defining a time window for processing events. C-Sparql is able to incrementally re-materialise the input data, using partial static background knowledge. The novel idea behind C-Sparql is the author's contribution to add an *expiration date* to each RDF triple in order to support fast deletion of events that are no longer valid. However, the use of time window frames for linking events prohibits the opportunity to link previous events with current or future events. Similarly, *Streaming SPARQL* [26] provides an extension of semantics and algebraic functions of SPARQL, which translates queries into logical algebra plans.

Etalis is an open-source engine that is able to detect and report changes over events in near real time, by combining both static and streaming knowledge. It incorporates the *Etalis Language for Events* (ELE) and *Event Processing SPARQL* (Ep-Sparql) [10]. The core of Etalis is implemented in Prolog and incorporates the fundamentals of logic programming: an event is modeled by ELE using logic facts and Prolog-style rules. In addition, Ep-Sparql was used to assist real-time complex event detection. In contrast to *C-Sparql*, using this framework, the user is able to define time windows in the past.

A novel approach in the area of query processing over Linked Stream Data is C-Quels [108]. In this work, the authors propose a white-box approach for querying stream data efficiently. To this end, they define and use techniques such as query optimisation, caching and indexing. Similarly, Instans [165] (which is based on the Rete-algorithm) is able to process streams of RDF data and cache the data after the processing is over. Moreover, Instans is the only approach that supports the simultaneous processing of SPARQL queries, where the immediate results of a query can be used from other queries once stored. Additionally, another SPARQL query extension language is described in [200], where the authors propose τ-SPARQL, which, combined with an index structure for temporal intervals, achieves better runtime performance.

Most of these approaches focus on extending the semantics and functions of SPARQL. To the best of our knowledge, the only SPARQL extension that incorporates Allen's Interval Algebra is T-Sparql [74]. T-Sparql is a temporal extension of SPARQL using the multi-temporal RDF database model of [73], using similar design characteristics as [192]. To query an event KB, T-Sparql enhances the `FILTER` field in order to identify links between mono-dimensional temporal data. T-Sparql utilizes operators that explicitly define the *bf*, *eq*, *ov*, *mt* and *dur* relations.

3.3 Semantic Similarities for Link Discovery

Over the past few years, semantic similarities have been utilized extensively in OM [166, 201]. In this context, concepts in two ontologies O_1 and O_2 are often matched based on a third ontology, e.g., WordNet. This ontology can be viewed as a background knowledge source or a mediating ontology [37]. Frameworks such as AgreementMaker [38], Zhishi.links [150] and RuleMiner [148] utilize semantic similarities in this way to improve structural matching in ontology level. While these enhancements have a positive effect on their resource-level matching, no resource linking tool, to the best of our knowledge, has used semantic similarities directly and shown an improvement of the overall linking results. [125] compares the effect of a predefined set of combinations of string and semantic similarities for label comparison and suggests that semantic similarities do not improve the F-measure of the instance matching task. Our results suggest the contrary by showing that dataset-specific combinations of measure actually can achieve this fit.

There have been a number of surveys on semantic similarities such as [27], which performs a comparison of five different measures. [16] presents an overview of different existing measures and finally, [125] compares the effect of a predefined set of combinations of string and semantic similarities for label comparison. In contrast to our study, they do not observe or evaluate the effect of syntactic and semantic similarity within LD, but perform comparisons in class level.

3.4 Fast Execution of Link Specifications

3.4.1 Executing Link Specifications under Time Constraints

The problem of identifying appropriate LSs using ML techniques has been explored in various previous papers. The approach in [85] focuses on generating LSs using genetic programming. Many approaches propose alternative methods to address the problem of minimizing the need for training examples. Eagle [141], Raven [140], and Agp [44] are some of the approaches that have used active learning in order to maintain a high level of accuracy by requesting less labeled examples by training. Wombat [188] uses only positive examples for finding accurate LSs. Additionally, unsupervised methods and tools such as Paris [144], Euclid [142] and KnoFuss [145] require no training data but are based on specific assumptions about the characteristics of the matching pairs. Except Euclid, the other approaches have been proven not to be deterministic. To the best of our knowledge the only method that focuses on identifying informative link candidates for training is Coala [143]. To this end, Coala uses the correlations between unlabelled examples. Based on their result evaluation, the authors proved the importance of identifying informative links to be used as training examples for LSs learning. Frameworks such as Febrl [32] and Marlin [9] rely on models such as Support Vector Machines and Decision Trees to identify appropriate classifiers. The main drawback of all previous approaches is that they assume that there are no time limitations in acquiring the desired data. To the best of our knowledge, we are the first to address the problem of partial-recall LD.

There has been significant work in the field of learning with Refinement Operator (RO), especially in the area of Inductive Logic Programming. The most notable work has been the Model Inference System [185], where Saphiro describes how a refinement operator can be used to adapt a hypothesis to a sequence of examples. Additionally, [208] presents a complete but improper upward refinement operator, and proves that locally finite, complete and proper refinement operators for unrestricted search spaces ordered by θ-subsumption do not exist. One of the most significant papers published in this field is [112], which analyzed the desirable properties of refinement operators and later extended to possible abstract property combinations in [113].

Finally, refinement operators have been introduced in many Description Logic languages such as $\mathcal{ALER}$ [15], $\mathcal{ALN}$ [57] and $\mathcal{EL}$ [111].

3.4.2 Planning for Link Discovery

The task of execution optimization using runtime approximations in LD frameworks is similar to the task of efficient query execution in database systems. Efficient and scalable data management has been of central importance in database systems [72]. Over the past few years, there has been extensive work on query optimization in databases, which is based on statistical information about relations and intermediate results [189]. The author of [29] gives an analytic overview regarding the procedure of query optimization and the different approaches used at each step of the process.

A novel approach in this field is presented in [84], in which the proposed approach introduces the concept of parametric query optimization. In this work, the authors provide the necessary formalization of the aforementioned concept and present a set of experiments in which they used the buffer size as parameter. In order to minimize the total cost of generating all possible alternative execution plans, they used a set of randomized algorithms. In a similar manner, the authors of [202] introduce the idea of Multi-Objective Parametric query optimization (MPQ), where the cost of a plan is associated with multiple cost functions and each cost function is associated with various parameters. Their experimental results show however that the MPQ method performs an exhaustive search of the solution space which makes this approach computationally inefficient.

Another set of approaches in the field of query optimization have focused on creating dynamic execution plans. Dynamic planning is based on the idea that the execution engine of a framework knows more than the planner itself. Therefore, information generated by the execution engine is used to re-evaluate the plans generated by the optimizer. There have been a vast amount of approaches for dynamic query optimization, such as query scrambling for initial delays [206], dynamic planning in compile-time [34], genetic programming [23], cost-based and heuristic optimizers [93], adaptive query operators [87] and re-ordering of operators [13].

The main difference between the task at hand and query optimization for databases is that databases can store elaborate statistics on the data they contain and use these to optimize their execution plan. LD frameworks do not have such statistics available when presented with a novel LS as they usually have to access remote data sources. Thus, planners must rely on statistics that can be computed efficiently while reading the data. Moreover, planning approaches for LD also has to rely on generic approximations for the costs and selectivity of plans. Still, we reuse the concepts of selectivity, rewriting and planning as known from query optimization in databases. Condor is the first dynamic planner for LD and dynamic approaches for query planning were the inspiration for the work presented herein.

3.5 Link Discovery Frameworks and Tools

Since this work is a contribution to the research area of LD, we begin by giving an overview of existing frameworks that were developed to assist scalable and time-efficient solutions towards LD. These frameworks commonly rely on scalable approaches for computing simple and complex specifications. For example, a lossless framework that uses blocking is Silk [86], a tool relying on rough index pre-matching and multi-dimensional blocking. KnoFuss [145] on the other hand implements classical blocking approaches derived from databases. These approaches are not guaranteed to achieve result completeness. Zhishi.links [150] is another framework that scales (through an indexing-based approach) but is not guaranteed to retrieve all links. Codi uses a sampling-based approach to compute anchor alignments to reduce its runtime [82]. The completeness of results is guaranteed by the Limes framework, which combines time-efficient algorithms such as Ed-Join [219] and PPJoin+ [221] with a set-theoretical combination strategy. The execution of LSs in Limes is carried out by means of the Canonical [137], Helios [138] and Condor [66] planners. Given that Limes was shown to outperform Silk in [138], we chose to implement our approaches throughout Chapters 5- 7 in Limes.

4 Aegle: An Efficient Approach for the Generation of Allen Relations

Preamble This chapter is based on [68], which was one of the first papers that focused on the scalability of time relations in LD. The author has co-designed, implemented and evaluated the algorithm presented herein, and co-wrote the said paper.

In this Chapter, we consider the efficient computation of temporal relations between events. Hence, we assume that each of the resources in S and T considered in the subsequent portion of this Chapter describes an event v with a beginning time denoted $b(v)$ and an end time denoted $e(v)$. Note that we assume that $b(v) < e(v)$ throughout this Chapter.

4.1 Allen's Interval Algebra

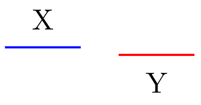
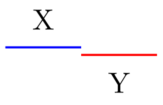
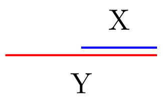
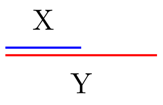
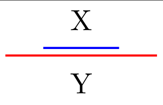
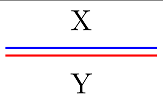
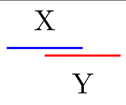
Allen's Interval Algebra [8] is a widely known time interval calculus, which provides a set of 13 "distinct, exhaustive, and qualitative" relations between time intervals [8]. Table 4.1 illustrates this set of relations and shows a set of six relations between two time intervals X and Y, their corresponding symbols along with the symbols of their inverse relation. The *equal* relation is symmetric.

4.2 Link Discovery between Events

An event can be modelled as a time interval because we assume that its description always includes a begin time property and an end time property, Thus, an event instance s can be described as pair of time points $(b(s), e(s))$, where $b(s) < e(s)$. Formally, computing the temporal relations between events can thus be reduced to computing the following mappings $\mathcal{M}$:

- if $Rel = bf$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) < b(t)) \wedge (b(s) < e(t)) \wedge (e(s) < b(t)) \wedge (e(s) < e(t))\}$
- if $Rel = bfi$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) > b(t)) \wedge (b(s) > e(t)) \wedge (e(s) > b(t)) \wedge (e(s) > e(t))\}$
- if $Rel = mt$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) < b(t)) \wedge (b(s) < e(t)) \wedge (e(s) = b(t)) \wedge (e(s) < e(t))\}$
- if $Rel = mti$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) > b(t)) \wedge (b(s) = e(t)) \wedge (e(s) > b(t)) \wedge (e(s) > e(t))\}$

Table 4.1: Allen's Interval Algebra

Relation	Notation	Inverse	Illustration
X before Y	$bf(X,Y)$	$bfi(X,Y)$	
X meets Y	$mt(X,Y)$	$mti(X,Y)$	
X finishes Y	$fin(X,Y)$	$fini(X,Y)$	
X starts Y	$st(X,Y)$	$sti(X,Y)$	
X during Y	$dur(X,Y)$	$duri(X,Y)$	
X equal Y	$eq(X,Y)$	$eq(X,Y)$	
X overlaps with Y	$ov(X,Y)$	$ovi(X,Y)$	

- if $Rel = fin$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) > b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) = e(t)\}$
- if $Rel = fini$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) < b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) = e(t)\}$
- if $Rel = st$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) = b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) < e(t)\}$
- if $Rel = sti$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) = b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) > e(t)\}$
- if $Rel = dur$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) > b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) < e(t)\}$
- if $Rel = duri$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) < b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) > e(t)\}$
- if $Rel = eq$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) = b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) = e(t)\}$
- if $Rel = ov$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) < b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) < e(t)\}$
- if $Rel = ovi$, then $\mathcal{M} = \{(s,t) \in S \times T : (b(s) > b(t)) \wedge (b(s) < e(t)) \wedge (e(s) > b(t)) \wedge e(s) > e(t)\}$

4.3 The Aegle Approach

The main goal of Aegle (Allen's intErval alGebra for Link discovEry) is to efficiently compute all Allen's interval relations, as described in Section 4.1, between two sets of atomic events. The main principle underlying this work is that we can reduce the computation of the 13 relations to the computation and combinations of a mere 8 atomic relations, and thus reduce the overall computation time of Allen relations. We use this insight to devise a means to compute all interval relations efficiently. To do so we reduce Allen's relations to re-usable atomic relations, which can then be computed efficiently. We further combine the results of these atomic relations to compute Allen's relations. While doing so, we ensure 100% accuracy in retrieving all possible Allen relations between resources in the given sets of resources S and T.

4.3.1 Atomic Temporal Relations

The main idea behind our approach is to represent each relation of Table 4.1 as a Boolean combination of atomic relations. By computing each of the atomic relations only once and only if needed, we can decrease the overall runtime of the computation of a given set of Allen relations.

As described in Section 4.2, each atomic event s can be described using two time points $b(s)$ and $e(s)$. To compose the atomic interval relations, we define all possible binary relations between the begin and end points of two event resources $s = (b(s), e(s))$ and $t = (b(t), e(t))$, as follows:

- Atomic relations between $b(s)$ and $b(t)$:
  - $BB^1(s,t) \Leftrightarrow (b(s) < b(t))$
  - $BB^0(s,t) \Leftrightarrow (b(s) = b(t))$
  - $BB^{-1}(s,t) \Leftrightarrow (b(s) > b(t)) \Leftrightarrow \neg(BB^1(s,t) \vee BB^0(s,t))$
- Atomic relations between $b(s)$ and $e(t)$:
  - $BE^1(s,t) \Leftrightarrow (b(s) < e(t))$
  - $BE^0(s,t) \Leftrightarrow (b(s) = e(t))$
  - $BE^{-1}(s,t) \Leftrightarrow (b(s) > e(t)) \Leftrightarrow \neg(BE^1(s,t) \vee BE^0(s,t))$
- Atomic relations between $e(s)$ and $b(t)$:
  - $EB^1(s,t) \Leftrightarrow (e(s) < b(t))$
  - $EB^0(s,t) \Leftrightarrow (e(s) = b(t))$
  - $EB^{-1}(s,t) \Leftrightarrow (e(s) > b(t)) \Leftrightarrow \neg(EB^1(s,t) \vee EB^0(s,t))$
- Atomic relations between $e(s)$ and $e(t)$:
  - $EE^1(s,t) \Leftrightarrow (e(s) < e(t))$
  - $EE^0(s,t) \Leftrightarrow (e(s) = e(t))$
  - $EE^{-1}(s,t) \Leftrightarrow (e(s) > e(t)) \Leftrightarrow \neg(EE^1(s,t) \vee EE^0(s,t))$

4.3.2 Complex Temporal Relations

Out of Table 4.1, we can derive how each of Allen's relations can be reduced to a Boolean combination of a subset of the relations above, as follows:

- $bf(s,t) \Leftrightarrow BB^1(s,t) \wedge BE^1(s,t) \wedge EB^1(s,t) \wedge EE^1(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $e(s) < b(t) \Rightarrow b(s) < b(t)$,
  2. $e(s) < b(t) \Rightarrow b(s) < e(t)$ (by virtue of 1.) and
  3. $e(s) < b(t) \Rightarrow e(s) < e(t)$.

  Hence $bf(s,t) = EB^1(s,t)$.

- $bfi(s,t) \Leftrightarrow BB^{-1}(s,t) \wedge BE^{-1}(s,t) \wedge EB^{-1}(s,t) \wedge EE^{-1}(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) > e(t) \Rightarrow b(s) > b(t)$,
  2. $b(s) > e(t) \Rightarrow e(s) > b(t)$ (by virtue of 1.) and
  3. $b(s) > e(t) \Rightarrow e(s) > e(t)$.

  Hence $bfi(s,t) = BE^{-1} = \neg(BE^1(s,t) \vee BE^0(s,t))$.

- $mt(s,t) \Leftrightarrow BB^1(s,t) \wedge BE^1(s,t) \wedge EB^0(s,t) \wedge EE^1(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$ and, by virtue of the transitivity of $<$, we get

  1. $e(s) = b(t) \Rightarrow b(s) < b(t)$,
  2. $e(s) = b(t) \Rightarrow e(s) < e(t)$ and
  3. $e(s) = b(t) \Rightarrow b(s) < e(t)$ (by virtue of 1.).

  Hence $mt(s,t) = EB^0(s,t)$.

- $mti(s,t) \Leftrightarrow BB^{-1}(s,t) \wedge BE^0(s,t) \wedge EB^{-1}(s,t) \wedge EE^{-1}(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) = e(t) \Rightarrow b(s) > b(t)$,
  2. $b(s) = e(t) \Rightarrow e(s) > e(t)$ and
  3. $b(s) = e(t) \Rightarrow e(s) > b(t)$ (by virtue of 1.).

  Hence $mti(s,t) = BE^0(s,t)$.

- $fin(s,t) \Leftrightarrow BB^{-1}(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^0(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) > b(t) \Rightarrow e(s) > b(t)$ and
  2. $e(s) = e(t) \Rightarrow b(s) < e(t)$

  Hence $fin(s,t) = \{EE^0(s,t) \wedge BB^{-1}(s,t)\} = \{EE^0(s,t) \wedge \neg(BB^0(s,t) \vee BB^1(s,t))\}$.

- $fini(s,t) \Leftrightarrow BB^1(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^0(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) < b(t) \Rightarrow b(s) < e(t)$ and

2. $e(s) = e(t) \Rightarrow e(s) > b(t)$

Hence $fini(s,t) = \{BB^1(s,t) \wedge EE^0(s,t)\}$.

- $st(s,t) \Leftrightarrow BB^0(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^1(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) = b(t) \Rightarrow b(s) < e(t)$ and
  2. $e(s) < e(t) \Rightarrow e(s) > b(t)$

  Hence $st(s,t) = \{BB^0(s,t) \wedge EE^1(s,t)\}$.

- $sti(s,t) \Leftrightarrow BB^0(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^{-1}(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) = b(t) \Rightarrow e(s) > b(t)$ and
  2. $b(s) = b(t) \Rightarrow b(s) < e(t)$

  Hence $sti(s,t) = \{BB^0(s,t) \wedge EE^{-1}(s,t)\} = \{BB^0(s,t) \wedge \neg(EE^0(s,t) \vee EE^1(s,t))\}$.

- $dur(s,t) \Leftrightarrow BB^{-1}(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^1(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) > b(t) \Rightarrow e(s) > b(t)$ and
  2. $e(s) < e(t) \Rightarrow b(s) < e(t)$

  Hence $dur(s,t) = \{EE^1(s,t) \wedge BB^{-1}(s,t)\} = \{EE^1(s,t) \wedge \neg(BB^0(s,t) \vee BB^1(s,t))\}$.

- $duri(s,t) \Leftrightarrow BB^1(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^{-1}(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $e(s) > e(t) \Rightarrow e(s) > b(t)$ and
  2. $b(s) < b(t) \Rightarrow b(s) < e(t)$

  Hence $duri(s,t) = \{BB^1(s,t) \wedge EE^{-1}(s,t)\} = \{BB^1(s,t) \wedge \neg(EE^0(s,t) \vee EE^1(s,t))\}$.

- $eq(s,t) \Leftrightarrow BB^0(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^0(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $e(s) = e(t) \Rightarrow e(s) > b(t)$ and
  2. $b(s) = b(t) \Rightarrow b(s) < e(t)$

  Hence $eq(s,t) = \{BB^0(s,t) \wedge EE^0(s,t)\}$.

- $ov(s,t) \Leftrightarrow BB^1(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^1(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $b(s) < b(t) \Rightarrow b(s) < e(t)$.

  Hence $ov(s,t) = \{BB^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^1(s,t)\} = \{(BB^1(s,t) \wedge EE^1(s,t)) \wedge \neg(EB^0(s,t) \vee EB^1(s,t))\}$.

- $ovi(s,t) \Leftrightarrow BB^{-1}(s,t) \wedge BE^1(s,t) \wedge EB^{-1}(s,t) \wedge EE^{-1}(s,t)$. Now given that $b(s) < e(s)$ and $b(t) < e(t)$, and by virtue of the transitivity of $<$, we get

  1. $e(s) > e(t) \Rightarrow e(s) > b(t)$.

Hence $ovi(s,t) = \{BB^{-1}(s,t) \wedge BE^{1}(s,t) \wedge EE^{-1}(s,t)\} = \{(BE^{1}(s,t) \wedge \neg(BB^{0}(s,t) \vee BB^{1}(s,t))) \wedge \neg(EE^{0}(s,t) \vee EE^{1}(s,t))\}$.

Clearly, we hence only need to compute the 8 atomic relations $EB^0, EB^1, EE^0, EE^1, BB^0, BB^1, BE^0$ and BE^1 to be able to generate all of Allen's relations. In the following, we explicate our approach to computing these 8 relations efficiently.

4.3.3 The Aegle Algorithm

Given a set $\mathcal{AIR}$ of Allen relations that are to be computed, the basic idea behind our approach is to begin by detecting the subset of the 8 atomic relations that needs to be computed, and to compute each of these relations exactly once. Algorithm 1 describes how the idea is implemented. Our approach, Aegle, takes two sets of events, S and T, and the set $\mathcal{AIR}$ as input. The algorithm returns a set of mappings M. Each mapping corresponds to exactly one of the relations in $\mathcal{AIR}$.

We begin by initialising the final set of mappings M in line 1 and the map $\mathcal{A}$ in line 2. $\mathcal{A}$ includes the labels of atomic relations as keys and their corresponding mapping as values. During the first step of our algorithm, for each $rel \in \mathcal{AIR}$, Aegle retrieves the set of required atomic relations in line 4 by calling the function $getAtmRelations(rel)$. This function is responsible for retrieving the set of atomic relation labels required to compute rel based on the rules defined in Section 4.3.2. For each required *atomicRel* of the current *rel*, the algorithm checks if the mapping is already computed (line 7). If not, it invokes the function *computeAtmRelation* to compute the appropriate atomic relations in line 8 and places the resulting mapping along with the atomic relation label in $\mathcal{A}$. Then, it retrieves the mapping from $\mathcal{A}$ and places it in the $atomics(rel)$ map needed to compute the mapping of *rel*. Each Allen's relation described in $\mathcal{AIR}$ constructs its own $atomics(rel)$ map, which has the labels of the requisite atomic relations as keys and their corresponding mappings as values. Finally, the algorithm computes the mapping *tempM* of *rel* by calling the function *computeRelation* (line 11) and adds the resulting set of links in M (line 12).

The time-critical portion of the execution lies in the computation of atomic relations. The idea underlying our approach to computing these relations is that one can reduce their computation to the problem of finding pairs of matching elements in two sorted lists. For example, to compute BB^0, one needs to (1) sort the list of elements of S and T according to the time at which they began (guaranteed time complexity: $O(|S|\log|S|)$ resp. $O(|T|\log|T|)$), (2) search for the elements of the smaller set in the larger set ($O(\min(|S|,|T|)\log(\max(|S|,|T|)))$. This leads to an overall complexity of $O(n\log n)$. The complexity is the same for the computation of all relations.

To illustrate the main procedure of Algorithm 1 (lines 3- 12), consider $\mathcal{AIR} = \{st, sti\}$ as an example. The other relations are computed analogously. In line 8 of Algorithm 1, Aegle calls *computeAtmRelations* in order to generate the mappings for the required atomic relations for *rel*, where $rel = st$ and $requiredRelations = BB^0, EE^1$. Since $\mathcal{A}$ is empty and the condition in line 7 holds, Algorithm 1 will call the function *computeAtmRelation* for BB^0 and then for EE^1.

For BB^0, Algorithm 2 describes the necessary steps to compute the mapping of BB^0. To begin with, Algorithm 2 invokes the function *orderByDate* for the source S and the target T datasets, to order both complex event resources using the property *beginDate*. Algorithm 4 illustrates the procedure of ordering a complex event S given the value of a property *dateType* - in this case *beginDate*. The main idea of this function is to assert each atomic event $s \in S$ to the appropriate time-bucket, given its *dateType* value. *orderByDate* returns a map that has the unique *dateType* values of the input KB S as keys and the set of events that correspond to each *dateType* as values. Once *sources* and *targets* are retrieved (lines 2, 3 resp. of Algo-

Algorithm 1: Aegle

```
Input: source $S$, target $T$, set of Allen relations $\mathcal{AIR}$
Output: Set of mappings $\mathcal{M}$
1  $M \longleftarrow \emptyset$
2  $\mathcal{A} \longleftarrow \emptyset$
3  foreach $rel \in \mathcal{AIR}$ do
4      $requiredRelations \longleftarrow getAtmRelations(rel)$
5      $atomics(rel) \longleftarrow \emptyset$
6      foreach $atomicRel \in requiredRelations$ do
7          if $\mathcal{A}$ does not contain $atomicRel$ then
8              $X \longleftarrow computeAtmRelation(atomicRel, S, T)$
9              $\mathcal{A}.put(atomicRel, X)$
10         $atomics(rel).put(atomicRel, \mathcal{A}.get(atomicRel))$
11     $tempM \longleftarrow computeRelation(atomics(rel))$
12     $M.add(tempM)$
13 Return $M$
```

Algorithm 2: $computeAtmRelations(atomicRel, S, T)$ for $atomicRel = BB^0$

```
Output: mapping of $BB^0$ $AM$
1  $AM \longleftarrow \emptyset$
2  $sources \longleftarrow orderByDate(S, \text{beginDate})$
3  $targets \longleftarrow orderByDate(T, \text{beginDate})$
4  $AM \longleftarrow mapEvents(sources, targets, \text{concurrent})$
5  Return $AM$
```

Algorithm 3: $computeAtmRelations(atomicRel, S, T)$ for $atomicRel = EE^1$

```
Output: mapping of $EE^1$ $AM$
1  $AM \longleftarrow \emptyset$
2  $sources \longleftarrow orderByDate(S, \text{endDate})$
3  $targets \longleftarrow orderByDate(T, \text{endDate})$
4  $AM \longleftarrow mapEvents(sources, targets, \text{predecessor})$
5  Return $AM$
```

Algorithm 4: $orderByDate(S, dateType)$

```
Output: $O$
1  foreach $s \in S$ do
2      $timeStamp \longleftarrow s.getDate(dateType)$
3      $tempO \longleftarrow \emptyset$
4      if $O$ contains $timeStamp$ then
5          $tempO \longleftarrow O.get(timeStamp)$
6      $tempO \longleftarrow tempO \cup s$
7      $O.put(timeStamp, tempO)$
8  Return $O$
```

Algorithm 5: *mapEvents*(*sources*, *targets*, *eventType*)

```
   Output: mapped events $Events$
1  $Events \longleftarrow \emptyset$
2  foreach $sourceTimeStamp \in sources$ do
3      if $eventType == concurrent$ then
4          $tempT \longleftarrow targets.get(sourceTimeStamp)$
5      else
6          $tempT \longleftarrow targets.getHigher(sourceTimeStamp)$
7      if $tempT != \emptyset$ then
8          foreach $s \in sources.get(sourceTimeStamp)$ do
9              $Events.put(s, tempT)$
10 Return $Events$
```

Algorithm 6: *computeRelation*(*atomics*) for *st*

```
   Output: mapping $M$
1  $M \longleftarrow \emptyset$
2  foreach $s \in atomics.get(BB^0)$ do
3      $M1 \longleftarrow atomics.get(BB^0).get(s)$
4      $tempEE1 \longleftarrow atomics.get(EE^1)$
5      if $tempEE1$ contains $s$ then
6          $M2 \longleftarrow tempEE1.get(s)$
7          $M.put(s, M1 \cap M2)$
8  Return $M$
```

rithm 2), *computeAtmRelations* calls the function *mapEvents* using the label *concurrent*, which is responsible for matching each source event s with the set of target events with the same $b(s)$. In the function *mapEvents* (Algorithm 5), for each source event s that belongs to a time-bucket with time-stamp *sourceTimeStamp*, the algorithm retrieves the appropriate subset of target events that have the same time-stamp (line 4), if any (line 7). Then, it constructs a mapping between each s and the matching set of target events (line 9). Finally, the mapping is returned to Algorithm 1 and placed in $\mathcal{A}$ in line 9.

To continue, Algorithm 1 again calls *computeAtmRelations*, since the mapping of EE^1 is not also contained in $\mathcal{A}$, following the procedure described in Algorithm 3. For EE^1, *computeAtm-Relations* orders S and T by invoking the *orderByDate* function that orders the event sources using the *endDate* property. Once both *sources* and *targets* are retrieved (lines 2, 3 resp. of Algorithm 3), *mapEvents* will be called with $eventType = predecessor$, in order to match each source $s \in S$ with the target events that were terminated after the source event s ended (line 6 of Algorithm 5). Finally, the mapping is returned to the main algorithm and is placed in $\mathcal{A}$ in line 9.

Once both mappings of BB^0 and EE^1 are retrieved and placed in $atomics(rel)$, Algorithm 1 calls *computeRelation* for *st*. Algorithm 6 illustrates the procedure of computing *st*. For each source event s, the algorithm retrieves the set of targets with the same $b(s)$ from the *atomics* set (line 3). Then, Algorithm 6 checks if a set of targets exist with *endDate* higher than $e(s)$ (line 5). If the condition holds, then *computeRelation* retrieves the aforementioned set of targets (line 6) and, based on the equation in Section 4.3.2, it computes the intersection between the two sub-sets of targets. The procedure is performed for each source instance and the final mapping M is returned in Algorithm 1 and placed in M.

Then, Aegle computes the *sti* relation, following the steps described above. However, since $sti(s,t) = \{BB^0(s,t) \wedge \neg(EE^0(s,t) \vee EE^1(s,t))\}$, the algorithm will only have to compute EE^0 and retrieve the mappings for BB^0 and EE^1.

4.4 Evaluation

4.4.1 Evaluation Questions

The aim of our evaluation was to address the following questions:

- Q_1: Does the reduction of Allen relations to 8 atomic relations influence the overall runtime of the approach?
- Q_2: How does Aegle perform in comparison with the state of the art in terms of time efficiency?

To the best of our knowledge, only one other Link Discovery framework implements an approach for the discovery of temporal relations. In [191], the blocking approach underlying Silk is extended to deal with spatio-temporal data. We thus compared our approach with the Silk LD framework.

4.4.2 Evaluation Datasets

We evaluated our approach on two different sets of datasets (see Table 4.2 for their characteristics):

- The first set of datasets (*3KMachines, 30KMachines, 300KMachines*) was created by generating synthetic event data using information obtained from real logs generated by production machinery. To this end, we retrieved 30,000 events from production machines, which covered a full day of event generation.[1] Then, we computed the probability that an event began or ended at any given point in time. Finally, we constructed our synthetic datasets by generating a fixed number of events that maintained the probability of an event beginning or ending at a particular point in time. For both *3KMachines* and *300KMachines* datasets, the distribution of unique begin and end date combinations found within the resources of the *30KMachines* dataset was the same as in the *30KMachines* dataset. The number of events within each unique time frame was decreased and increased by 10% for *3KMachines* and *300KMachines* resp.
- The second set of datasets (*3KQueries, 30KQueries, 300KQueries*) was obtained by collecting real event data from query logs of triple stores exposed on the Web. The data was retrieved from the SPARQL endpoint of the *LSQ* project [172].[2] For each dataset, we performed a *SPARQL* query against the *LSQ* endpoint and obtained a set of events from a set of consecutive days.

All 6 datasets were pre-processed in order to comply with implementation requirements of both Aegle and Silk.

[1]The source of the events cannot be disclosed due to legal reasons.

[2]More information can be found at `http://aksw.github.io/LSQ/`

Table 4.2: Characteristics of data sets. Size stands for the number of events contained in the dataset.

Log Type	Dataset name	Size	Unique $b(s)$	Unique $e(s)$
Machinery	*3KMachines*	3,154	960	960
	30KMachines	28,869	960	960
	300KMachines	288,690	960	960
Query	*3KQueries*	3,888	3,636	3,638
	30KQueries	30,635	3,070	3,070
	300KQueries	303,991	184	184

4.4.3 Experimental Setup

As evaluation measure, we computed the *runtime* of each of the atomic relations, the *time* required by our implementation to perform the *computeRelation* for each Allen Relation (Algorithm 6) and the *total runtime* required for computing all 13 relations. For Silk, we measured the time it required to compute each Allen relation.[3]

We set the value of Silk's block size to 1 ms.[4] Each temporal relation implemented in Silk was given a maximum runtime of 6 hours. We will use the symbol *NA* to signify that a run did not terminate within 6 hours. For the sake of comparison, we also implemented a naive *baseline* for the *eq* relation. This naive implementation performs an exhaustive comparison of the events of S and T to compute *eq*. For each experiment, we linked each data source with itself, i.e., we set $S = T$. All experiments for all implementations were carried out on the same 20-core Linux Server running *OpenJDK* 64-Bit Server 1.8.0_74 on Ubuntu 14.04.4 LTS on Intel(R) Xeon(R) CPU E5-2650 v3 processors clocked at 2.30GHz. Each experiment was run on exactly one core using 64 GB of RAM. We implemented Aegle using Java 1.8.0_60 and the sorting algorithm described in *orderByDate* (Algorithm 4) was performed using the *MergeSort* algorithm [100] as implemented in Java 1.8.0_60 with a guaranteed time complexity $O(n \log n)$.

4.4.4 Experimental Results

To address Q_1, we computed the execution runtime of all 8 atomic relations as described in Section 4.3. Table 4.4 shows the runtimes of the atomic relations as well as the total runtime required to run the full set of atomic relations. For our largest dataset *300KQueries*, our approach needed only 84.83 s to compute all atomic relations. The maximum required runtime is reached on the *300KMachines* dataset, while our algorithm needs approximately 7 min. As expected, the atomic relations which rely on equality (i.e., BB^0, BE^0, EB^0, EE^0) require less time than the rest of the atomic relations.

Another interesting observation derived from Table 4.4 is the relation between the size of the data, the number of the unique $b(s)$ and $e(s)$ among the event sources and the execution runtime of each relation. In the *Machines* datasets, the distribution of beginning and end times is equal among the different sizes of data. As expected by virtue of the complexity of our approach, the total runtimes grow in accordance with $O(n \log n)$ as the data increases. From *Query* datasets, we notice that the number of unique $b(s)$ and $e(s)$ has a significant impact on the

[3]To measure this time, we contacted the author of [191], who informed us that measuring the duration of the "Match Task" was the way to measure the runtime of his approach.

[4]We contacted the authors of Silk's temporal relation extension and were informed that this setting should return the best results.

Table 4.3: Total runtime of Allen Relations for all datasets for Aegle and Silk. All runtimes are presented in seconds.

Log Type	Dataset Name	Total Runtime		
		Aegle	Aegle *	Silk
Machine	*3KMachines*	11.26	5.51	294.00
	30KMachines	1,016.21	437.79	29,846.00
	300KMachines	189,442.16	78,416.61	*NA*
Query	*3KQueries*	26.94	17.91	541.00
	30KQueries	988.78	463.27	33,502.00
	300KQueries	211,996.88	86,884.98	*NA*

runtime of our approach. For example, even though *300KQueries* includes 10 times more data than *30KQueries*, *30KQueries* has a significantly higher number of unique $b(s)$ and $e(s)$ than *300KQueries*. Hence, Aegle requires 15 secs less for *300KQueries* than for the *30KQueries* dataset. The benefits of our implementation can be noticed clearly when comparing Aegle with the *baseline* (see Table 4.6). For the *eq* relation, we see that Aegle is 470 times faster than the brute-force approach. We can thus answer Q_1 by stating that (1) both the number of unique events and the distribution of events across time have a significant influence on the overall runtime and (2) our approach improves the overall runtime of the computation of Allen relations significantly.

Table 4.4: Execution runtime of all 8 atomic relations for all datasets. All runtimes are presented in seconds.

Log Type	Dataset Name	BB^0	BB^1	BE^0	BE^1	EB^0	EB^1	EE^0	EE^1	Total Runtime
Machine	*3KMachines*	0.02	0.41	0.02	0.41	0.02	0.41	0.02	0.42	1.73
	30KMachines	0.19	5.55	0.19	5.51	0.19	5.48	0.18	5.49	22.78
	300KMachines	2.70	95.55	2.14	92.26	3.39	115.66	2.13	94.4	408.23
Query	*3KQueries*	0.03	2.93	0.03	3.04	0.02	2.89	0.03	2.90	11.87
	30KQueries	0.19	24.5	0.19	26.28	0.21	23.85	0.19	23.80	99.22
	300KQueries	2.52	12.11	1.98	12.57	3.89	25.41	1.93	24.42	84.83

Tables 4.3, 4.5 and 4.6 provide us with the insights necessary to answer Q_2. They show clearly that Aegle outperforms Silk on all datasets in terms of time efficiency while achieving 100% precision and recall, i.e., while computing all the links that can be found. Therefore, our idea proves to be beneficial and time-efficient for the task of linking temporal data of various sizes.

In more detail, Aegle requires $211,996.88\,s$ to run the complete computation of Allen relations on our largest dataset (*300KQueries*), whereas Silk is unable to produce full results for any of the relations within the time frame of $280,800\,s$ (3.25 days). *30KQueries* is the largest dataset for which Silk was able to produce links for the given time limit. Here, we observe that Aegle is more than 33 times faster than Silk. Furthermore, Table 4.5 suggests that the most costly operations are carried out for inverse relations. However, by relying on the semantics of

Table 4.5: Execution runtime of the 13 Allen Relations for all datasets for Aegle and Silk and *baseline*. The runtimes reported for Aegle are the times required to perform the set operations necessary to compute each relation. The overall runtimes (i.e., computation of required sets plus times for set operations) are presented in Table 4.6. All runtimes are presented in seconds.

		Machine			Query		
Relation	Approach	*3KMachines*	*30KMachines*	*300Machines*	*3KQueries*	*30KQueries*	*300KQueries*
bf	Aegle	0.00	0.00	0.05	0.00	0.00	0.03
	Silk	22.00	2,511.00	*NA*	43.00	2,794.00	*NA*
bfi	Aegle	1.52	127.37	27,103.19	2.37	127.37	32,023.10
	Silk	24.00	2,547.00	*NA*	42.00	2,961.00	*NA*
mt	Aegle	0.00	0.00	0.03	0.00	0.00	0.00
	Silk	23.00	2,219.00	*NA*	41.00	2,466.00	*NA*
mti	Aegle	0.00	0.00	0.03	0.00	0.00	0.00
	Silk	23.00	2,290.00	*NA*	44.00	2,584.00	*NA*
fin	Aegle	0.73	77.88	13,775.31	1.18	70.53	16,280.24
	Silk	23.00	2,306.00	*NA*	41.00	2,531.00	*NA*
$fini$	Aegle	0.42	47.07	7,837.04	0.62	40.04	8,600.89
	Silk	23.00	2,305.00	*NA*	43.00	2,535.00	*NA*
st	Aegle	0.21	29.48	4,849.29	0.34	22.70	5,796.87
	Silk	21.00	2,166.00	*NA*	40.00	2,613.00	*NA*
sti	Aegle	0.74	76.14	14,063.02	1.19	69.69	16,270.20
	Silk	21.00	2,226.00	*NA*	43.00	2,533.00	*NA*
dur	Aegle	1.14	125.20	24,094.20	1.84	107.60	26,213.64
	Silk	24.00	2,363.00	*NA*	41.00	2,546.00	*NA*
$duri$	Aegle	1.20	125.04	24,083.00	1.83	108.50	26,149.58
	Silk	23.00	2,293.00	*NA*	41.00	2,476.00	*NA*
eq	Aegle	0.01	0.40	45.01	0.00	0.06	344.10
	Silk	23.00	2,250.00	*NA*	41.00	2,473.00	*NA*
	baseline	2.05	171.10	23,436.30	3.15	196.09	31,452.54
ov	Aegle	1.70	182.04	35,244.48	2.68	163.16	38,165.31
	Silk	22.00	2,181.00	*NA*	39.00	2,487.00	*NA*
ovi	Aegle	1.87	202.80	37,939.27	3.02	179.90	42,068.13
	Silk	22.00	2,189.00	*NA*	42.00	2,503.00	*NA*

Allen relations, we can refrain from computing inverse relations and have them inferred by any forward or backward chaining system. The results under Aegle* in Table 4.3 show that overall, the total runtime for computing the seven Allen relations bf, mt, fin, st, dur, eq and ov amounts to less than half of Aegle's runtime.

To conclude our answer for Q_2, we studied what would happen if we computed each of the Allen relations individually, i.e., we ran 13 experiments where we set $\mathcal{AIR}$ to contain exactly one Allen relation. We used this setting to allow for a fine-grained comparison of our runtimes with Silk's. The results of this experiment are shown in Table 4.6. Overall, we manifestly outperform Silk, even when computing each of the Allen relations on its own. This suggests that our core implementation for the computation of atomic relations is superior to the generic blocking scheme followed by Silk. This is especially clear when looking at the results on large datasets in more detail.

For *30KQueries* for example, Silk needs 2,473 seconds while Aegle only needs 0.45. The answer to Q_2 is hence that Aegle outperforms the state of the art in all our experimental settings. Note that the total runtime of a relation is increased by the number of atomic relations involved in its computation when computed using Aegle. As a result, Aegle needs more time for the *ovi* relation (which is derived by combining 5 atomic relations) than for *eq* (2 atomic relations).

Table 4.6: Execution runtime of all Allen Relations if computed individually. All runtimes are presented in seconds.

		Machine			Query		
Relation	Approach	*3KMachines*	*30KMachines*	*300Machines*	*3KQueries*	*30KQueries*	*300KQueries*
bf	Aegle	0.41	5.48	115.71	2.89	23.86	25.44
	Silk	22.00	2,511.00	*NA*	43.00	2,794.00	*NA*
bfi	Aegle	1.95	133.42	27,197.59	5.58	153.84	32,037.64
	Silk	24.00	2,547.00	*NA*	42.00	2,961.00	*NA*
mt	Aegle	0.02	0.19	3.42	0.02	0.21	3.89
	Silk	23.00	2,219.00	*NA*	41.00	2,466.00	*NA*
mti	Aegle	0.02	0.20	2.17	0.03	0.19	1.98
	Silk	23.00	2,290.00	*NA*	44.00	2,584.00	*NA*
fin	Aegle	1.20	84.18	13,875.70	4.20	95.67	16,296.80
	Silk	23.00	2,306.00	*NA*	41.00	2,531.00	*NA*
fini	Aegle	0.85	53.74	7,934.73	3.57	64.78	8,614.93
	Silk	23.00	2,305.00	*NA*	43.00	2,535.00	*NA*
st	Aegle	0.66	35.23	4,946.39	3.29	46.70	5,823.81
	Silk	21.00	2,166.00	*NA*	40.00	2,613.00	*NA*
sti	Aegle	1.20	83.71	14,162.25	4.13	94.39	16,299.07
	Silk	21.00	2,226.00	*NA*	43.00	2,533.00	*NA*
dur	Aegle	2.10	138.55	24,286.85	7.69	156.87	26,252.70
	Silk	24.00	2,363.00	*NA*	41.00	2,546.00	*NA*
duri	Aegle	2.15	138.47	24,275.08	7.67	157.75	26,188.04
	Silk	23.00	2,293.00	*NA*	41.00	2,476.00	*NA*
eq	Aegle	0.05	0.79	49.84	0.05	0.45	348.51
	Silk	23.00	2,250.00	*NA*	41.00	2,473.00	*NA*
	baseline	2.05	171.10	23,436.30	3.15	196.09	31,452.54
ov	Aegle	2.96	199.73	35,553.48	11.42	236.87	38,231.15
	Silk	22.00	2,181.00	*NA*	39.00	2,487.00	*NA*
ovi	Aegle	3.16	222.27	38,226.32	11.97	257.59	42,121.68
	Silk	22.00	2,189.00	*NA*	42.00	2,503.00	*NA*

5 Semantic Similarities for Link Discovery

Preamble This chapter is based on [67]. It is the first approach to the efficient and scalable computation of four well-known edge-counting semantic similarities for LD. The author co-designed, implemented and evaluated the algorithm presented herein, and co-wrote the said paper.

5.1 Notations

In this section, we begin by defining the notation necessary to model a lexical vocabulary as a Directed Acyclic Graph (DAG). Then, based upon these definitions, we introduce the edge-counting semantic similarities that we use in this work.

5.1.1 Lexical Vocabulary as Directed Acyclic Graph

We define a lexical vocabulary as a DAG $G = (V, E)$, where:

- The set of vertices V is a set of concepts c_i, were each c_i stands for a set of synonyms. We denote $|V|$ with n_V.
- $E \subseteq V \times V$ is a set of directed edges $e_{jk} = (c_j, c_k)$. We denote $|E|$ with n_E.
- The *edge* e_{jk} stands for the hypernymy relations from a parent concept c_j to a child concept c_k. We write $c_j \rightarrow c_k$ and say that c_j is a hypernym of c_k. We also define the hyponymy relation as a directed relation from a child concept c_k to a parent concept. We write $c_j \leftarrow c_k$ and we say that c_j is a hyponym of c_k. Hypernymy and hyponymy are transitive.
- The node *root* is the unique node of dictionary that has no parent concept.
- A *leaf concept* c_i is a concept node with no children concepts.
- A concept is a *common subsumer* of c_1 and c_2 (denoted $cs(c_1, c_2)$) iff that concept is a hypernym of both c_1 and c_2.
- The *least common subsumer* (LSO) of c_1 and c_2 (denoted $lso(c_1, c_2)$) is “the most specific concept which is an ancestor of both c_1 and c_2” [218].

- We define the *directed path* from c_1 to c_2 via a common subsumer $cs(c_1, c_2)$ as: $path(c_1, c_2) = \{c_1 \leftarrow c_i \leftarrow \ldots \leftarrow cs(c_1, c_2) \rightarrow c_j \rightarrow \ldots \rightarrow c_2 : i, j, k \in \mathbb{N}, i, j, k \leq n_v\}$. Note that there are multiple $path(c_1, c_2)$ between two concepts.

- $len(c_1, c_2)$ is *the length of the shortest* $path(c_1, c_2)$ between two concepts c_1 and c_2. Note that *len* defines a metric. Hence, it is symmetric and abides by the triangle inequality, i.e., $len(c_1, c_2) \leq len(c_1, c_3) + len(c_2, c_3)$ for any $(c_1, c_2, c_3) \in V^3$.

- We define $depth_m(c_i)$ as the length of the shortest path between *root* and c_i. Analogously, $depth_M(c_i)$ is defined as the maximum $depth(c_i)$. We set $D = \max_{c \in V} depth_M(c)$.

Note that the following holds:

- $depth_m(root, c_i) = len(root, c_i)$

- $depth_m(lso(c_1, c_2)) \leq min(depth_m(c_1), depth_m(c_2))$

- $depth_M(lso(c_1, c_2)) \leq min(depth_M(c_1), depth_M(c_2))$

- (triangle inequality) $|len(root, c_1) - len(root, c_2)| \leq len(c_1, c_2) \Leftrightarrow |depth_m(c_1) - depth_m(c_2)| \leq len(c_1, c_2)$

5.1.2 Edge-Counting Semantic Similarities

We present normalized versions of common edge-counting similarities.

Shortest Path Similarity

The Shortest Path (Shortest Path) similarity [156] between two concepts c_1 and c_2 is defined as Shortest Path
$(c_1, c_2) = 2D - len(c_1, c_2)$. Given that Shortest Path$(c_1, c_2) \in [0, 2D]$, we used the normalized formulation of Shortest Path, i.e.,

$$\text{Shortest Path}(c_1, c_2) = \frac{2D - len(c_1, c_2)}{2D}. \tag{5.1}$$

Leacock and Chodorow

The Leacock and Chodorow metric (LCH- <u>L</u>eacock and <u>CH</u>odorow) takes both the path between two concepts and the depth of the hierarchy into consideration [109]:

$$\text{LCH}(c_1, c_2) = -\log\left(\frac{len(c_1, c_2)}{2D}\right). \tag{5.2}$$

where LCH$(c_1, c_2) \in (0, \log(2D)]$ for $c_1 \neq c_2$. The normalized version of LCH is hence

$$\text{LCH}(c_1, c_2) = \begin{cases} 1 \text{ if } c_1 = c_2 \\ \frac{-\log\left(\frac{len(c_1, c_2)}{2D}\right)}{\log(2D)} \text{ else.} \end{cases} \tag{5.3}$$

Wu Palmer

The Wu and Palmer similarity (Wu Palmer) takes both the path between two concepts and the depth of LSO into consideration [218]:

$$\text{Wu Palmer}(c_1, c_2) = \frac{2depth_M(lso(c_1, c_2))}{2depth_M(lso(c_1, c_2)) + N_1 + N_2} \tag{5.4}$$

where $N_1 = len(lso(c_1, c_2), c_1)$ and $N_2 = len(lso(c_1, c_2), c_2)$. This similarity function is normalized.

Li

The Li et al. metric (Li) is another take on using the path between two concepts and their LSO to define a similarity [118]:

$$\text{Li}(c_1, c_2) = e^{-\alpha len(c_1,c_2)} \frac{e^{\beta depth(lso(c_1,c_2))} - e^{-\beta depth(lso(c_1,c_2))}}{e^{\beta depth(lso(c_1,c_2))} + e^{-\beta depth(lso(c_1,c_2))}} \tag{5.5}$$

where $\text{Li}(c_1, c_2) \in (0, 1)$. We set $depth(lso(c_1, c_2)) = depth_M(lso(c_1, c_2))$, since the original specification does not state which $depth(lso(c_1, c_2))$ to use.

5.2 Approach

To show that semantic similarities can indeed help achieve better F-measures than the state of the art, we first had to devise a scheme to reduce the runtimes of semantic similarities within the framework of LD. In hECATE (Edge-Counting semAntic similariTies for Link DiscovEry), we provide such a formal framework for edge-counting similarity, which we use to reduce the runtime of our subsequent experiments.

Fundamentally, hECATE aims to compute the set $\mathcal{M}^* = \{(s, t) \in S \times T : m(s, t, p_s, p_t) \geq \theta\}$, where m represents an edge-counting semantic similarity. To achieve this goal, the approach makes use of upper bounds, which can be derived from the formulation of this family of measures. Take the Shortest Path similarity for example: for any two concepts c_1 and c_2, $\text{Shortest Path}(c_1, c_2) \geq \theta$ implies $len(c_1, c_2) \leq 2D(1 - \theta)$. Formally, this means that we can discard all comparisons of pairs (c_1, c_2) with $len(c_1, c_2) > 2D(1 - \theta)$ without compromising the computation of $\mathcal{M}^*$. Note that the computation of $len(c_1, c_2)$ can be carried out online or offline, which affects the total runtime of our approach as discussed in Section 5.3. As similar bounds can be derived for the other edge-counting measures, hECATE generalizes the computation of $\mathcal{M}^*$ for edge-counting semantic similarities by using Algorithm 7.

5.2.1 hECATE

The hECATE approach takes (1) two sets of resources, S and T, (2) an atomic LS $L = ((m(p_s, p_t), \theta)$, where m is one of the four semantic similarities described in Section 5.1.2, and (3) a lexical vocabulary structured as DAG (Section 5.1.1) as input. Our goal is to compute the mapping $M = [[L]]$ (see line 29 of Algorithm 7). For each pair (s, t), hECATE retrieves and pre-processes the property values for p_s resp. p_t (lines 6, 5). The pre-processing consists of tokenizing and extracting all stop-words from the objects of the triples (s, p_s, o_s) and (t, p_t, o_t). In order to include a pair (s, t) in M, the algorithm compares each set of source tokens from o_s ($sTokens$) to each set of target tokens of o_t ($tTokens$). The pair of objects (o_s, o_t) with the highest similarity abiding by the bounds derived for each measure (see paragraph above for an example) is used to compute the similarity between s and t and then, the algorithm decides whether or not this pair should be added to M.

To do so, for each token $sToken \in sTokens$, we find the $tToken \in tTokens$ that is most similar to each $sToken$ (lines 12- 21). First, the algorithm checks if $sToken$ and $tToken$ have been compared before (line 12) by checking the set *mapSim*. If the tokens are being compared for the first time (line 13), the algorithm checks if the tokens are identical and assigns the value of 1 to $TTSim$. Or, it calls the function $compare(sToken, tToken, VDAG)$, which compares the corresponding sets of concepts obtained from the input $VDAG$ (line 16). In both cases, the *mapSim* is updated accordingly. Then, $TTSim$ is compared to the maximum token-to-token similarity and $maxTTSim$ is updated. The procedure continues until the highest similarity between the current $sToken$ and a $tToken$ is found or the loop stops if the $maxTTSim$ is equal to 1 (line 21).

The algorithm aggregates the highest similarities $maxTTSim$ of all $sToken \in sTokens$ (line 23), takes their average (line 24), compares with the current highest similarity, and replaces it accordingly (line 25). The loops continues by comparing all pairs of ($sTokens$,$tTokens$). In cases where the *similarity* of a set of ($sTokens$,$tTokens$) is equal to 1, the comparison of the set of tokens is terminated. Once the final *maxSimilarity* is found, our algorithm decides if the pair (s, t) can be added to the final mapping M (line 29).

The key behind hECATE lies in the token comparison algorithm - $compare(sToken, tToken, VDAG)$ (line 16 of Algorithm 7). So, for a pair of tokens $(sToken, tToken)$, we retrieve the set of concepts they belong to in the $VDAG$. If both sets of concepts are not empty, we compare each source $sCon$ with target concept $tCon$ and define the maximum similarity of two tokens as the highest similarity of the corresponding concept pair. To do so, Algorithms 8 and 9 describe our implementation of the edge-counting semantic similarities as part of the hECATE algorithm. For all similarities, we first retrieve the set of all hypernym paths of each concept to the root of the $VDAG$ (lines 2, 3 of Algorithm 8, and lines 1, 2 of Algorithm 9). The $getPaths(concept, VDAG)$ algorithm traverses the $VDAG$ by utilizing the hypernym relation. It starts from the *concept* node and explores as far as possible along each path before backtracking.

For Shortest Path and LCH, we retrieve the maximum depth D found in the $VDAG$ (line 1), the $len(sCon, tCon)$ (line 4), and we proceed in calculating the corresponding similarity as described in Equation 5.1 and 5.2 resp. in line 5. Algorithm 10 describes the process of identifying the $len(sCon, tCon)$ by considering the set hypernym paths of the concepts. The algorithm begins by iterating over both paths hp_1 and hp_2 simultaneously, from top to bottom, until they do not share any common nodes (line 5). Then, it proceeds in calculating the length of the newly found $path(sCon, tCon)$ (line 7), as the number of concepts that the two paths do not have in common. Finally, the algorithm compares the size of the newly found path with the existing one and replaces it accordingly (line 8). This procedure is repeated for all pairs of hypernym paths until the $len(sCon, tCon)$ is found.

For Wu Palmer and Li, we retrieved the depth of LSO between $sCon$ and $tCon$ ($depth(lso$ $(sCon, tCon))$, and N_1 and N_2 (line 3) by calling the function $getLSO$ (hps_1, hps_2). Algorithm 11 describes the procedure of identifying the aformentioned values by considering the set of hypernym paths of the concepts. The algorithm begins by iterating over both paths hp_1 and hp_2 simultaneously, in a similar manner as in Algorithm 10. Then, it proceeds in calculating the length of the newly found $path(sCon, tCon)$ (line 8) and replaces the values of $depth(lso(sCon, tCon))$, N_1 and N_2 iff there is no LSO found, or the new LSO is located deeper in the hierarchy than the current LSO, or the new LSO has the same depth with the current LSO but the new $path(sCon, tCon)$ is smaller (line 5). This procedure is repeated for all pairs of hypernym paths until the $depth_M$ $(lso(sCon, tCon)$ and the corresponding N_1 and N_2 are found. Once Algorithm 11 returns, we proceed in calculating the corresponding similarity as

Algorithm 7: hECATE($S, T, L, VDAG$)

```
Input: source KB $S$, target KB $T$, a LS $L = ((m(p_s, p_t), \theta)$ and a vocabulary DAG $VDAG$
Output: a mapping $M$
1  $M \leftarrow \emptyset$
2  foreach $(s, t) \in S \times T$ do
3      $maxSimilarity \leftarrow 0$
4      $simMap \leftarrow \emptyset$
5      $newTargets \leftarrow preprocess(t, p_t)$
6      foreach $sTokens \in preprocess(s, p_s)$ do
7          foreach $tTokens \in newTargets$ do
8              $similarity \leftarrow 0$
9              foreach $sToken \in sTokens$ do
10                 $maxTTSim \leftarrow 0$
11                 foreach $tToken \in tTokens$ do
12                     $TTSim \leftarrow checkSimilarity(simMap, sToken, tToken)$
13                     if $TTSim == -1$ then
14                         if $sToken == tToken$ then
15                             $TTSim \leftarrow 1$
16                         else
17                             $TTSim \leftarrow compare(sToken, tToken, VDAG)$
18                         $update(simMap, sToken, tToken, TTSim)$
19                     if $TTSim > maxTTSim$ then
20                         $maxTTSim \leftarrow TTSim$
21                     if $maxTTSim == 1$ then
22                         $break$
23                 $similarity \leftarrow similarity + maxTTSim$
24             $similarity \leftarrow similarity / sTokens.getSize()$
25             if $similarity > maxSimilarity$ then
26                 $maxSimilarity \leftarrow similarity$
27             if $maxSimilarity == 1$ then
28                 $break$
29     if $maxSimilarity \geq \theta$ then
30         $M \leftarrow M \cup (s, t)$
31 Return $M$
```

described in Equation 5.4 and 5.5 resp. in line 4.

5.2.2 Indexing

Our first extension of hECATE is based on the idea of pre-computing and storing a set of values that are used often in our algorithm. For edge-counting similarites, these are the hypernym paths. Consequently, the extension hECATE-I (hECATE with Indexing) of hECATE precomputes all hypernym paths for all concepts included in the $VDAG$, using the $getPaths(concept, VDAG)$

Algorithm 8: *compare*$(sCon, tCon,\ VDAG)$ for Shortest Path or LCH

```
  Input: source concept $sCon$, target concept $tCon$, and a vocabulary DAG $VDAG$
  Output: a $similarity$ value
1 $D \leftarrow VDAG.getMaxDepth(sCon)$
2 $hps_1 \leftarrow getPaths(sCon, VDAG)$
3 $hps_2 \leftarrow getPaths(tCon, VDAG)$
4 $minLength \leftarrow getMinLength(hps_1, hps_2)$
5 Return $computeSimilarity(D, minLength)$
```

Algorithm 9: *compare*$(sCon, tCon,\ VDAG)$ for Wu Palmer or Li

```
  Input: source concept $sCon$, target concept $tCon$, and a vocabulary DAG $VDAG$
  Output: a $similarity$ value
1 $hps_1 \leftarrow getPaths(sCon, VDAG)$
2 $hps_2 \leftarrow getPaths(tCon, VDAG)$
3 $depth, N_1, N_2 \leftarrow getLSO(hps_1, hps_2)$
4 Return $computeSimilarity(N_1, N_2, depth)$
```

function. Therefore, every time the *getPaths*$(concept, VDAG)$ is invoked (lines 2, 3 of Algorithm 8 and lines 1, 2 of Algorithm 9), instead of computing all the hypernym paths of *concept* on the fly, we retrieve them from an index.

5.2.3 Filtering

Our second extension of hECATE, hECATE-IF (hECATE with Indexing and Filtering), combines hECATE-I with the idea of minimizing unnecessary comparison between concepts by filtering out pairs of source and target concepts that do not satisfy a condition for each semantic similarity. The filtering is performed inside *compare*$(sToken, tToken, VDAG)$ for each pair of concepts $sCon$ and $tCon$. Given a semantic similarity, if a pair of concepts satisfies the corresponding filtering condition, then the algorithm proceeds normally as described in Section 5.2.1. This check is performed before line 2 in the *compare*$(sToken, tToken,\ VDAG)$ function for Shortest Path or LCH, and before line 1 in the *compare*$(sToken, tToken,\ VDAG)$ function for Wu Palmer or Li. If the condition is not met, then the algorithm does not compute the similarity between the two concepts.

For the Shortest Path similarity, two concepts will be considered for comparison if the following holds:

$$\begin{aligned} \textsc{Shortest Path}(c_1, c_2) \geq \theta &\Leftrightarrow \\ \frac{2D - len(c_1, c_2)}{2D} \geq \theta &\Rightarrow \\ |depth_m(c_1) - depth_m(c_2)| \leq 2D(1-\theta) & \end{aligned} \tag{5.6}$$

For the Wu Palmer similarity, the following must hold:

$$\begin{aligned}
WU(c_1, c_2) \geq \theta \Leftrightarrow \\
\frac{2depth_M(lso(c_1, c_2))}{2depth_M(lso(c_1, c_2)) + N_1 + N_2} \geq \theta \Leftrightarrow \\
2depth_M(lso(c_1, c_2)) \geq \theta(N_1 + N_2) + 2\theta depth_M(lso(c_1, c_2)) \Leftrightarrow \\
N_1 + N_2 \leq \frac{2depth_M(lso(c_1, c_2))(1-\theta)}{\theta} \Rightarrow \\
N_1 + N_2 \leq \frac{2min(depth_M(c_1), depth_M(c_2))(1-\theta)}{\theta}
\end{aligned} \tag{5.7}$$

Based on the triangle inequality and Section 5.1.2, Equation 5.7 can be written as:

$$\begin{aligned}
len(c_1, c_2) \leq \frac{2min(depth_M(c_1), depth_M(c_2))(1-\theta)}{\theta} \Rightarrow \\
|depth_m(c_1) - depth_m(c_2)| \leq \frac{2min(depth_M(c_1), depth_M(c_2))(1-\theta)}{\theta}
\end{aligned} \tag{5.8}$$

For the LCH similarity, two concepts will be considered for comparison if the following holds:

$$\begin{aligned}
\mathrm{LCH}(c_1, c_2) \geq \theta \Leftrightarrow \\
\frac{-log\frac{len(c_1,c_2)}{2D}}{log(2D)} \geq \theta \Leftrightarrow \\
\frac{log(2D) - log(len(c_1, c_2))}{log(2D)} \geq \theta \Leftrightarrow \\
1 - \frac{log(len(c_1, c_2))}{log(2D)} \geq \theta \Leftrightarrow \\
\frac{log(len(c_1, c_2))}{log(2D)} \leq (1-\theta) \Leftrightarrow \\
log(len(c_1, c_2)) \leq log(2D)(1-\theta) \Leftrightarrow \\
len(c_1, c_2) \leq 2^{log(2D)(1-\theta)} \Rightarrow \\
|depth_m(c_1) - depth_m(c_2)| \leq 2^{log(2D)(1-\theta)}
\end{aligned} \tag{5.9}$$

When considering the Li similarity, we make the following variable replacements for the sake of legibility: $x = depth_M(lso(c_1, c_2))$, $y = min(depth_M(c_1), depth_M(c_2))$ and $z = len(c_1, c_2)$. Then, two concepts will be considered for comparison, iff:

$$\begin{aligned}
\textsc{Li}(c_1, c_2) \geq \theta &\Leftrightarrow \\
e^{-\alpha z}\frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}} \geq \theta &\Leftrightarrow \\
e^{\alpha z} \leq \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}}\theta &\Leftrightarrow \\
e^{\alpha z} \leq \frac{\frac{(e^{2\beta x}-1)}{e^{\beta x}}}{\frac{(e^{2\beta x}+1)}{e^{\beta x}}\theta} &\Leftrightarrow \\
e^{\alpha z} \leq \frac{(e^{2\beta x} - 1)}{(e^{2\beta x} + 1)\theta} &\Rightarrow \\
e^{\alpha z} \leq \frac{(e^{2\beta y} - 1)}{(e^{2\beta y} + 1)\theta} &\Leftrightarrow \\
\alpha z \leq ln(e^{2\beta y} - 1) - ln\theta - ln(e^{2\beta y} + 1) &\Leftrightarrow \\
|depth_m(c_1) - depth_m(c_2)| \leq \frac{ln(e^{2\beta y} - 1) - ln\theta - ln(e^{2\beta y} + 1)}{\alpha}&
\end{aligned} \tag{5.10}$$

Based on Equations 5.6, 5.8, 5.9 and 5.10, each filtering condition requires the knowledge of $depth_m(sCon)$, $depth_M(sCon)$, $depth_m(tCon)$ and $depth_M(tCon)$. To do so, we traverse the $VDAG$ in the same manner as in the $getPaths(sCon, tCon, VDAG)$ function, obtain all paths for a concept and find the one with the maximum and minimum lengths.

Algorithm 10: $getMinLength(hps_1, hps_2)$

```
    Input: two sets of hypernym paths, $hps_1$ and $hps_2$
    Output: $len(sCon, tCon)$
1   $size \leftarrow MAX\_VALUE$
2   foreach $hp_1 \in hps_1$ do
3       foreach $hp_2 \in hps_2$ do
4           $l_1 \leftarrow 0$, $l_2 \leftarrow 0$
5           while $l_1 < hp_1.size() \wedge l_2 < hp_2.size() \wedge hp_1.get(l_1) == hp_2.get(l_2)$ do
6               $l_1 \leftarrow l_1 + 1$, $l_2 \leftarrow l_2 + 1$
7           $newSize \leftarrow hp_1.size() + hp_2.size() - 2l_1$
8           if $newSize < size$ then
9               $size \leftarrow newSize$
10  Return $size$
```

5.3 Evaluation

5.3.1 Evaluation Questions

The aim of our evaluation was to address the three research questions:

- Q_1: How do our strategies for improving the runtime of semantic similarities compare to each other w.r.t. runtime?
- Q_2: How do the different edge-counting semantic similarities compare w.r.t. runtime?
- Q_3: Can semantic similarities improve the F-measure of LD systems?

Algorithm 11: *getLSO*(hps_1, hps_2)

```
   Input: two sets of hypernym paths, $hps_1$ and $hps_2$
   Output: $depth_M(lso(sCon, tCon))$, $N_1$ and $N_2$
 1 $dLSO \leftarrow 0$, $N_1 \leftarrow 0$, $N_2 \leftarrow 0$
 2 foreach $hp_1 \in hps_1$ do
 3     foreach $hp_2 \in hps_2$ do
 4         $l_1 \leftarrow 0$, $l_2 \leftarrow 0$
 5         while $l_1 < hp_1.size() \wedge l_2 < hp_2.size() \wedge hp_1.get(l_1) == hp_2.get(l_2)$ do
 6             $l_1 \leftarrow l_1 + 1$
 7             $l_2 \leftarrow l_2 + 1$
 8         $newSize \leftarrow hp_1.size() + hp_2.size() - 2l_1$
 9         $oldSize \leftarrow N_1 + N_2$
10         if condition is met then
11             $dLSO \leftarrow l_1$
12             $N_1 \leftarrow hp_1.size() - l_1$
13             $N_2 \leftarrow hp_2.size() - l_2$
14 Return $dLSO, N_1, N_2$
```

5.3.2 Evaluation Datasets

We evaluated our approach against five benchmark data sets: Abt-Buy, Amazon-GP and DBLP-ACM described in [103], DailyMed-Drugbank (dubbed DM-DB) and DB-Movies described in [141]. We used WordNet[1] as a $VDAG$. WordNet is a "*large lexical database of English, where nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept*" [58].

Table 5.1: Characteristics of data sets

Dataset	**Source (S)**	**Target (T)**	$\|S\| \times \|T\|$	**Source Property**	**Target Property**
Abt-Buy	Abt	Buy	1.20×10^6	description	description
Amazon-GP	Amazon	Google	4.40×10^6	description	description
DBLP-ACM	ACM	DBLP	6.00×10^6	title	title
DM-DB	DailyMed	DrugBank	1.09×10^6	name	name
DB-Movies	DBpedia	LinkedMDB	1.11×10^6	title	title

5.3.3 Experimental Setup

To address Q_1 an Q_2, we conducted a set of experiments using the basic hECATE algorithm (dubbed hECATE-B) as a baseline as well as hECATE-I and hECATE-IF. All methods were implemented in LD framework Limes [137]. For hECATE-IF, we enhanced the Indexing extension described in Section 5.2.2, by additionally pre-computing and storing the $depth_m(c_i)$ and $depth_M(c_i)$ for every concept found in the $VDAG$ by traversing the graph in a BFS manner. For hECATE-B and hECATE-I, we created one atomic LS for each semantic similarity, where m was the name of the edge-counting similarity, $\theta = 0.1$. The source and target properties (p_s,

[1] `https://wordnet.princeton.edu/`

p_t resp.) for each dataset were derived from Table 5.1. For hECATE-IF, we used the same values for m, p_s and p_t as before, but θ was derived from the interval $[0.1, 1]$ with an increment step of 0.1, since the θ is given as a parameter to the filtering functions (Section 5.2.3). For each dataset, we performed the aforementioned LSs against 2^v instances from the source and target datasets, starting with $v = 2$, and we incremented v until all instances were covered.[2] We allowed each LS to run up to $2\,hrs$. Each experiment was executed 3 times and we present the average values.[3]

The second goal of this work was to evaluate edge-counting semantic similarities in LD in terms of accuracy. Consequently, for Q_3, we used the hECATE extension with the best runtime performance based on the results of Q_1 and we executed a set of experiments using 2 machine learning (ML) algorithms: Wombat [188] and Dragon [151]. We performed a 10-fold cross validation by allowing Wombat and Dragon to use as input only string similarities (StrSim), only semantic similarities (SmtSim) and a combination of both (StrSmtSim). We used the `levenshtein`, `cosine` and `qgrams` as string similarity measures, implemented in Limes [137]. For each dataset, we used all properties apart from those that corresponded to numeric values.

Wombat was configured as presented in [188], and Dragon was configured as presented in [151]. We used two termination criteria for Wombat: either a LS with F-measure of 1 was found, or a maximal refinement depth of 10 was reached. For string similarities, Wombat produced LSs with minimum θ value of 0.4, and for the semantic similarities, the minimum θ value was set to 0.7. Dragon terminated either when no new nodes were found, or when the maximum height = 3 of the decision tree was reached. All runtime experiments were carried out on a 64-core Linux Server running *OpenJDK* 64-Bit Server 1.8.0_121 on Ubuntu 16.04.3 LTS on Intel(R) Xeon(R) CPU E5-2698 v3 processors clocked at 2.30GHz.

5.3.4 Experimental Results

As expected, hECATE-B achieved the lowest performance compared to hECATE-I and hECATE-IF (Figure 5.1) in all datasets, except DM-DB. It is obvious that introducing the filtering and indexing extensions improves the runtime of all semantic similarities, making them more amenable for LD and scalable for larger datasets. Precisely, LCH's, Wu Palmer's and Shortest Path's runtimes improve by 71% and 57% on average when hECATE-I and hECATE-IF strategies are used resp. Li has the least improvement by 65% and 50%. Comparing the two extensions, in all datasets and for all semantic similarities, hECATE-I outperforms hECATE-IF by an average of 30%. A detailed analysis of the runtimes shows that even though hECATE-IF reduces the number of comparisons between semantically different concepts and thus the comparison time, the additional runtime cost of filtering creates an overhead that results in a worse total execution time than hECATE-I (Table 5.2). Regarding the DM-DB dataset, the only property for both source and target datasets, *name*, consists of only one value, which corresponds to the official name of a drug. That value can only be associated with one concept. As a result, introducing an indexing and/or filtering technique produces an unnecessary overhead. We can now clearly answer Q_1: our best strategy for semantic similarities is hECATE-I.

Our answer to Q_2 is based on Figure 5.1: the semantic similarity with the worst runtime performance is Li. For the DB-Movies dataset, we notice that hECATE-I requires 100K more token comparisons than the other similarities (Table 5.2). The reason behind this is twofold: (1) in our implementation, we check if the similarity of two tokens/concepts is 1 and stop any further comparison of tokens/concepts resp. (2) based on Equation 5.5, it is not possible to add this break point since the Li similarity can never be 1. However, based on Figure 5.1 and

[2]For the Amazon-Google dataset, the maximal value of v was set to 9.

[3]All results are available at `https://hobbitdata.informatik.uni-leipzig.de/hECATE/`.

Table 5.2: Number of concept comparisons produced for the DB-Movies dataset for hECATE-I and hECATE-IF. The number of comparisons for hECATE-B is the same as hECATE-I

Threshold		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Shortest Path	hECATE-I	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M
	hECATE-IF	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	**61.0M**	**51.4M**	**10.3M**
Wu Palmer	hECATE-I	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M
	hECATE-IF	61.8M	61.8M	61.8M	**61.7M**	**61.5M**	**60.3M**	**56.7M**	**44.7M**	**27.8M**	**10.3M**
LCH	hECATE-I	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M	61.8M
	hECATE-IF	61.8M	61.8M	61.8M	**61.4M**	**60.4M**	**56.5M**	**42.4M**	**42.4M**	**28.5M**	**28.5M**
Li	hECATE-I	61.9M	61.9M	61.9M	61.9M	61.9M	61.9M	61.9M	61.9M	61.9M	61.9M
	hECATE-IF	61.9M	**61.4M**	**59.9M**	**56.6M**	**51.5M**	**42.1M**	**28.3M**	**28.2M**	**10.0M**	**00.0M**

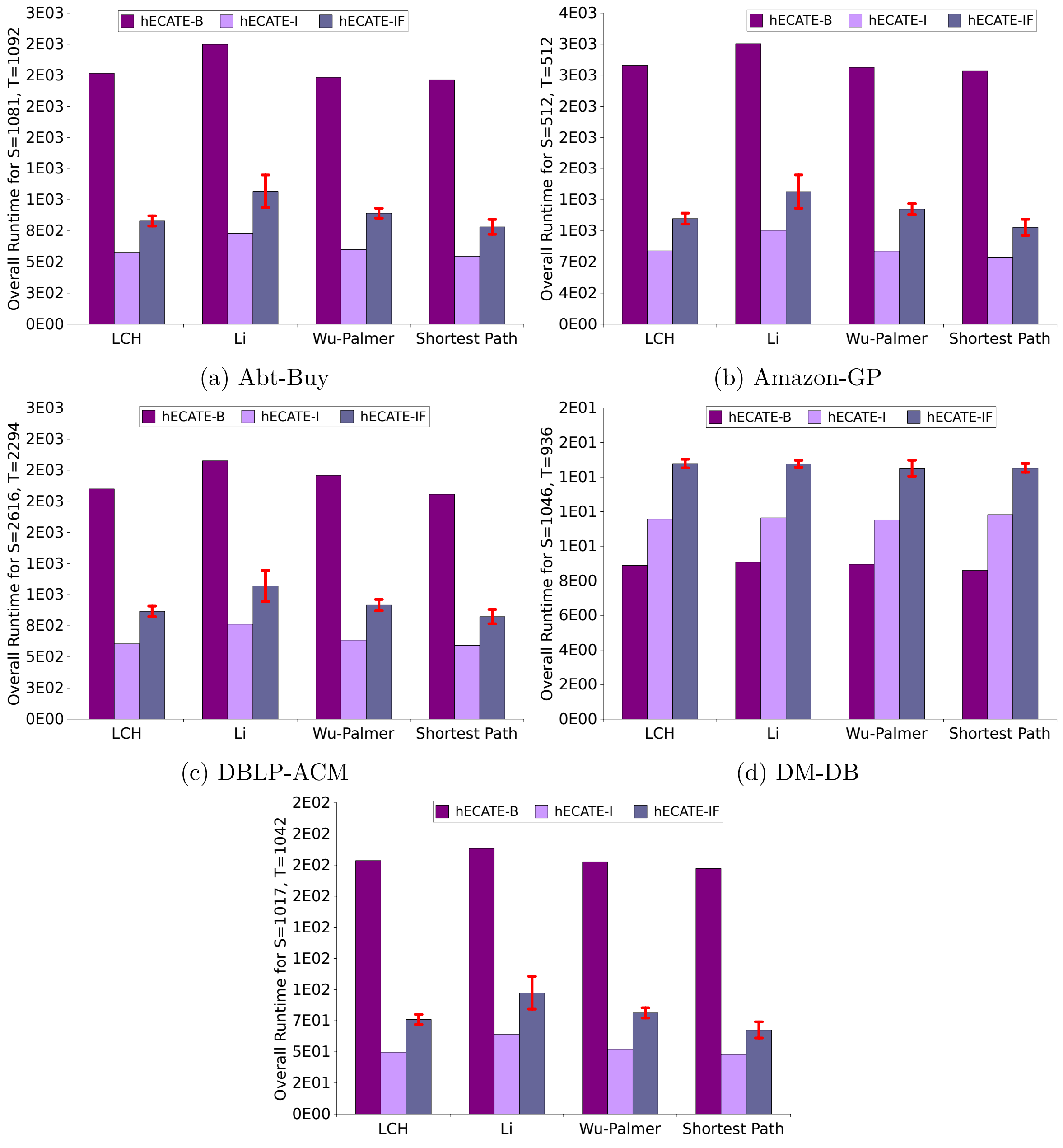
Table 5.2, Li's runtime shows a great improvement as the values of θ increase in relation to the other metrics. This justifies the fact that Li has the highest standard deviation, whereas Shortest Path, LCH and Wu Palmer are less influenced by the different values of θ. This answers Q_2: for all hECATE strategies, Shortest Path is the fastest similarity, whereas Li is the slowest.

To answer Q_3, we added the four edge-counting measures Li, Wu Palmer, Shortest Path, and LCH to the state-of-the-art algorithms Wombat and Dragon, where we evaluated their performance with and without string similarity using a ten-fold cross validation. We chose these two approaches because (1) they achieve state-of-the-art performance while being deterministic, (2) they are open-source, meaning our experiments can be easily reproduced and (3) they are able to generate complex LSs with any arbitrary number of measures. Table 5.3 shows the results of our experiments with machine-learning algorithms. Additionally, in Table 5.3, we reported the average F-measure score of the string-based LD algorithms Eagle [141], Euclid [136], J48 [80] reported by [151], and the Multilayer Perceptron (MultiPerc) classifier reported by [195]. Unfortunately, we were not able to include our semantic similarities in the said systems. However, solely based on the results of the string similarities, we observe that Wombat outperforms Eagle, Euclid, J48 and MultiPerc in 3 out of 5 datasets (Table 5.3).

Table 5.3: Average F-measure achieved by Wombat, Dragon, Euclid, Eagle, J48 and Multilayer Perception within a 10-fold cross validation setting. The semantic similarities use the hECATE-I strategy.

Algorithm	Wombat			Dragon			Euclid	Eagle	J48	MultiPerc
Similarities	StrSim	SmtSim	StrSmtSim	StrSim	SmtSim	StrSmtSim	StrSim	StrSim	StrSim	StrSim
Abt-Buy	0.65	0.65	**0.66**	0.51	0.02	0.10	0.00	0.56	0.43	0.43
Amazon-GP	0.71	0.60	**0.77**	0.64	0.06	0.05	0.71	0.73	0.41	0.36
DBLP-ACM	0.97	0.74	0.97	0.93	0.81	0.93	**0.98**	**0.98**	0.77	0.97
DM-DB	0.94	0.71	0.97	0.89	0.65	0.89	**1.00**	**1.00**	0.94	-
DB-Movies	1.00	0.73	**1.00**	0.93	0.80	0.93	0.98	0.99	0.84	-

Table 5.4 includes the maximum F-measure achieved by Wombat, Dragon, the Pessimistic and the Re-Weighted versions of the work presented at [97]. For comparison reasons, we conducted an additional set of experiments for Wombat and Dragon, following the experimental setup introduced in [97]. Each experiment was executed 7 times by randomly selecting 2% of the gold standard as training data and using the remaining 98% for testing. Since Pessimistic and Re-Weighted are not open-source, we were not able to find and alter the algorithms to include the edge-counting semantic similarities. However, the results obtained for string similarities presented in Table 5.4 show that Wombat outperforms both systems in



(a) Abt-Buy (b) Amazon-GP (c) DBLP-ACM (d) DM-DB (e) DB-Movies

Figure 5.1: Execution time results of applying hECATE-B, hECATE-I and hECATE-IF on all evaluation data. The y-axis shows the average runtime in seconds. For hECATE-IF, we also present the standard deviation among the different thresholds.

80% of cases.

Regarding the comparison of two ML systems that were able to support semantic similarities - the performance of Dragon remained the same in 60% of cases or even worsened; while adding semantic similarities to the Wombat algorithm improved its overall performance of the algorithm in 60% of cases by up to 6% F-measure absolute. As expected, this effect is most pronounced in datasets that rely on long textual descriptions such as Amazon-GP. A look into the specifications learned by Wombat suggests that this effect is due to the approach combining semantic and string similarities using an operator such as $\sqcup$, and learning the correct threshold for each of these measures. Improvement on the DM-DB datasets is achieved using the \ operator, not allowing semantically similar concepts to be matched together. This refutes current results (see [125]) and suggests that the refinement operators can combine semantic and string similarities in a way that improves the F-measure.

Table 5.4: Maximum F-measure achieved by Wombat, Dragon, Pessimistic and Re-Weighted achieved using 2% of the data for training over 7 iterations [97]. The semantic similarities use the hECATE-I strategy.

Algorithm	Wombat			Dragon			Pessimistic	Re-Weighted
Similarities	StrSim	SmtSim	StrSmtSim	StrSim	SmtSim	StrSmtSim	StrSim	StrSim
Abt-Buy	0.35	**0.39**	0.34	0.24	0.10	0.24	0.36	0.37
Amazon-GP	**0.53**	0.33	0.43	0.45	0.13	0.35	0.39	0.43
DBLP-ACM	0.91	0.55	0.91	0.90	0.66	0.90	0.93	**0.95**
DM-DB	0.94	0.71	**0.97**	0.94	0.71	0.96	-	-
DB-Movies	0.97	0.33	**0.97**	0.96	0.33	0.96	-	-

6 An Evaluation of Models for Runtime Approximation in Link Discovery

Preamble This chapter is based on [64] and is the first study of exponential and mixed models for the estimation of the planner runtimes. The author co-designed, implemented and evaluated the algorithm presented herein, and co-wrote the said paper.

6.1 Selection of Models for Runtime Approximation

Planners aim to estimate the cost of the leaves of a plan, i.e., the runtime of atomic LS. So far, linear models [138] have been used for this purpose but the appropriateness of other models has never been evaluated. Hence, in this work, we compare non-linear models with linear models to approximate the runtime of an atomic LS. Like in previous works, we follow a *sampling-based approach.* First, given a particular similarity measure m (e.g., Levenshtein) and an implementation of the said measure (e.g., *Ed-Join* [219]), we begin by collecting samples of runtimes for a given measure with varying values of $|S|$, $|T|$ and θ.[1] These samples can be regarded as the output of a function, which can predict the implementation runtime of m, for which we were given samples. The major question to be answered is hence, *what is the shape of the runtime evaluation function*?

We tried fitting functions of different shapes to the previously measured runtimes in order to compare their performance when planning the execution of link specifications. Formally, these functions are mappings $\psi : \mathbb{N} \times \mathbb{N} \times (0, 1] \mapsto \mathbb{R}$, whose value at $(|S|, |T|, \theta)$ is an approximation of the runtime for the link specification with these parameters. If $\vec{R} = (R_1, \ldots, R_n)$ are the measured runtimes for the parameters $\vec{S} = (|S_1|, \ldots, |S_n|)$, $\vec{T} = (|T_1|, \ldots, |T_n|)$ and $\vec{\theta} = (\theta_1, \ldots, \theta_n)$, then we constrain the mapping ϕ to be a local minimum of the L2-Loss:

$$E(\vec{S}, \vec{T}, \vec{\theta}) := \|\vec{R} - \phi(\vec{S}, \vec{T}, \vec{\theta})\|^2, \tag{6.1}$$

writing $\psi(\vec{S}, \vec{T}, \vec{\theta}) = (\psi(|S_1|, |T_1|, \theta_1), \ldots, \psi(|S_n|, |T_n|, \theta_n))$.

Within this paper, we consider the following parametrized families of functions:

$$\begin{aligned}
\psi_1(S, T, \theta) &= a + b|S| + c|T| + y\theta & (6.2)\\
\psi_2(S, T, \theta) &= \exp\left(a + b|S| + c|T| + y\theta + h\theta^2\right) & (6.3)\\
\psi_3(S, T, \theta) &= a + (b + c|S| + y|T| + h|S||T|) \exp\left(z\theta + x\theta^2\right) & (6.4)
\end{aligned}$$

[1] We also experimented with the number of trigrams contained in S and T but found that they do not affect the models we considered.

The parameters are then determined by

$$a^*, b^*, \dots = \arg\min E(\vec{S}, \vec{T}, \vec{\theta}, \vec{R})(a, b, \dots) \tag{6.5}$$

for some local minimum. In the case of ψ_1 and ψ_2 this problem is linear in nature and we solved it using the pseudo-inverse of the associated Vandermonde matrix. For ψ_3 we used the Levenberg-Marquardt Algorithm [130] for nonlinear least squares problems, using 1 as an initial guess for all parameters.

We chose ψ_1 as the baseline linear fit. ψ_2 was the standard log-linear fit, except for the θ^2 term. We included this term during a grid search for polynomials to perform a log-polynomial fit. Higher orders of $|S|$ or $|T|$ or θ did not contribute to a better fit. ψ_3 can be interpreted as an interpolation of ψ_1 and ψ_2 with a constant offset a.

To exemplify our approach for ψ_2, assume we have measured $\vec{S} = (458, 458, 358, 58), \vec{T} = (512, 404, 317, 512)$ and $\vec{\theta} = (0.5, 0.9, 0.6, 0.7)$. Inserting into Equation 6.1 and taking the logarithm, one arrives at the optimization problem

$$\min_{a,b,c,y,h} \| \begin{pmatrix} 1 & 458 & 512 & 0.5 & 0.5^2 \\ 1 & 458 & 404 & 0.9 & 0.9^2 \\ 1 & 358 & 317 & 0.6 & 0.6^2 \\ 1 & 58 & 512 & 0.7 & 0.7^2 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ y \\ h \end{pmatrix} - \begin{pmatrix} \log(67) \\ \log(4) \\ \log(4) \\ \log(1) \end{pmatrix} \|^2$$

The solution to this least squares problem also is the unique solution of its normal equations:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 458 & 458 & 358 & 58 \\ 512 & 404 & 317 & 512 \\ 0.5 & 0.9 & 0.6 & 0.7 \\ 0.5^2 & 0.9^2 & 0.6^2 & 0.7^2 \end{pmatrix} \begin{pmatrix} 1 & 458 & 512 & 0.5 & 0.5^2 \\ 1 & 458 & 404 & 0.9 & 0.9^2 \\ 1 & 358 & 317 & 0.6 & 0.6^2 \\ 1 & 58 & 512 & 0.7 & 0.7^2 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ y \\ h \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 & 1 & 1 \\ 458 & 458 & 358 & 58 \\ 512 & 404 & 317 & 512 \\ 0.5 & 0.9 & 0.6 & 0.7 \\ 0.5^2 & 0.9^2 & 0.6^2 & 0.7^2 \end{pmatrix} \begin{pmatrix} \log(67) \\ \log(4) \\ \log(4) \\ \log(1) \end{pmatrix}$$

By multiplying and inverting matrices, we arrive at the linear equation

$$\begin{pmatrix} a \\ b \\ c \\ y \\ h \end{pmatrix} = \begin{pmatrix} 1 & 458 & 512 & 0.5 & 0.5^2 \\ 1 & 458 & 404 & 0.9 & 0.9^2 \\ 1 & 358 & 317 & 0.6 & 0.6^2 \\ 1 & 58 & 512 & 0.7 & 0.7^2 \end{pmatrix}^+ \begin{pmatrix} \log(67) \\ \log(4) \\ \log(4) \\ 0 \end{pmatrix},$$

where A^+ denotes the Moore-Penrose pseudo inverse of A [36]. Multiplying the matrices, we arrive at

$$\begin{pmatrix} a \\ b \\ c \\ y \\ h \end{pmatrix} = \begin{pmatrix} -1.028 \\ 0.009 \\ 0.010 \\ 9.821 \\ -9.053 \end{pmatrix}.$$

Thus we have found the coefficients of the fit function.

6.2 Evaluation

6.2.1 Evaluation Questions

Each of our experiments consisted of two phases: During the *training* phase, we trained each of the models independently. For each model, we computed the set of coefficients for each of the approximation models that minimized the Root-Mean-Square Error (RMSE) on the training data provided. The aim of the subsequent *test* phase was to evaluate the accuracy of the runtime estimation provided by each model and the performance of the currently best LD planner, Helios [138], when it relied on each of the three models for runtime approximations. Throughout our experiments, we used the algorithms *Ed-Join* [219] (which implements the Levenshtein string distance) and *PPJoin+* [220] (which implements the jaccard, overlap, cosine and trigrams string similarity measures) to execute atomic specifications. For the threshold θ, we used random values between 0.5 and 1.

The aim of our evaluation was to answer the following set of questions regarding the performance of the three models: *exp*, *linear* and *mixed*. [2]

- Q_1: How do our models fit each class separately?

  To answer this question, we began by splitting the source and target data of each of our datasets into two non-overlapping parts of equal size. We used the first half of each source and each target for training and the second half for testing.

  - *Training*: We trained the three models on each dataset. For each model, dataset and mapper, we a) selected 15 source and 15 target samples of random sizes from the first half of a dataset, and b) compared each source sample with each target sample 3 times. Note that we used the same samples across all models for the sake of fairness. Overall, we ran 675 training experiments to train each model on each dataset.
  - *Testing*: To test the accuracy of each model, we ran the corresponding algorithm (*Ed-Join* and *PPJoin+*) with a random threshold between 0.5 and 1 and recorded the real runtime of the approach and the runtimes predicted by our three models. Each approach was executed 100 times against the whole of the second half of the same dataset.

- Q_2: How do our models generalize across classes, i.e., can a model trained on data from one class be used to predict runtimes accurately on another class?

  - *Training*: We trained each model in the same manner as for Q_1 on exactly the same five datasets, with the sole difference that the samples were selected randomly from the whole dataset.
  - *Testing*: As in the previous series of experiments, we ran *Ed-Join* and *PPJoin+* with a random threshold between 0.5 and 1. Each of the algorithms was executed 100 times against the remaining four datasets.

- Q_3: How do our models perform when trained on a large dataset?

  - *Training*: We trained in the same fashion as to answer Q_1, with two differences: (1) we used 15 source and 15 random target samples of various sizes between $10,000$ and $100,000$, and (2) the target samples used to train our model came from English labels of DBpedia.

[2] For Q_1 and Q_2 we did not conduct experiments using the dataset derived from DBPedia's English labels, since it includes labels from multiple classes.

– *Testing*: We learned 100 LSs for each dataset using the unsupervised version of the Eagle algorithm [141]. We chose this algorithm because it was shown to generate meaningful specifications that return high-quality links in previous works. For each dataset, we ran the set of 100 specifications learned by Eagle on the given dataset by using each of the models during the execution, in combination with the Helios planning algorithm [138]. This algorithm was shown to outperform the canonical planner w.r.t. runtime while producing exactly the same results.

6.2.2 Evaluation Datasets

We evaluated the three runtime estimation models using six datasets. The first three are the benchmark datasets for LD dubbed Amazon-GP, DBLP-ACM and DBLP-Scholar described in [103]. We also created two larger additional datasets (MOVIES and VILLAGES, see Table 6.1) from the datasets DBpedia, LinkedGeoData (LGD) and LinkedMDB.[3] [4] The sixth dataset was the set of all English labels from DBpedia 2014. Table 6.1 describes the characteristics of the datasets and presents the properties used when linking the retrieved resources for the first four datasets. The mapping properties were provided to the LD algorithms underlying our results.

6.2.3 Experimental Set-Up

Throughout our experiments, we configured Eagle by setting the number of generations and population size to 20, mutation and crossover rates were set to 0.6. All experiments for all implementations were carried out on the same 20-core Linux Server running *OpenJDK* 64-Bit Server 1.8.0_74 on Ubuntu 14.04.4 LTS on Intel(R) Xeon(R) CPU E5-2650 v3 processors clocked at 2.30GHz. Each *train* experiment and each *test* experiment for Q_3 was repeated three times. As evaluation measure, we computed RMSE between the *expected* runtime and the average *execution* runtime required to run each LS. We report all three numbers for each model and dataset.

Table 6.1: Entity matching characteristics of datasets

Dataset	**Source (S)**	**Target (T)**	$\lvert S\rvert \times \lvert T\rvert$	**Source Property**	**Target Property**
Amazon-GP	Amazon	Google Products	4.40×10^6	product name description manufacturer price	product name description manufacturer price
DBLP-ACM	ACM	DBLP	6.00×10^6	title, authors venue, year	title authors venue, year
DBLP-Scholar	DBLP	Google Scholar	0.17×10^9	title, authors venue, year	title, authors venue, year
MOVIES	DBpedia	LinkedMDB	0.17×10^9	dbp:name dbo:director/dbp:name dbo:producer/dbp:name dbp:writer/dbp:name rdfs:label	dc2:title movie:director/movie:director_name movie:producer/movie:producer_name movie:writer/movie:writer_name rdfs:label
VILLAGES	DBpedia	LGD	6.88×10^9	rdfs:label dbo:populationTotal geo:geometry	rdfs:label lgdo:population geom:geometry/agc:asWKT

[3]`http://www.linkedmdb.org/`

[4]The new datasets as well as a description of how they were constructed are available at `https://hobbitdata.informatik.uni-leipzig.de/LIGER/newDatasets/`.

6.2.4 Experimental Results

To address Q_1, we evaluated the performance of our models when trained and tested on the same class. We present the results of this series of experiments in Table 6.2. For *PPJoin+* (in particular the trigrams measure), the *mixed* model achieved the lowest error when tested upon Amazon-GP and DBLP-Scholar, whereas the *linear* model was able to approximate the expected runtime with higher accuracy on the MOVIES and VILLAGES datasets. On average, the *linear* model was able to achieve a lower RMSE compared to the other two models. For *Ed-Join*, the *mixed* model outperformed *linear* and *exp* in the majority of datasets (DBLP-Scholar, MOVIES and VILLAGES) and obtained, on average, the lowest RMSE. As we observe in Table 6.2, for both measures, the *exp* model retrieved the highest error on average and is thus the model least suitable for runtime approximations. Especially for *Ed-Join*, *exp* had the worst performance in four out of the five datasets and retrieved the highest RMSE among the different test datasets for VILLAGES. This clearly answers our first question: the *linear* and *mixed* approximation models are able achieve the smallest error when trained on the class on which they are tested.

Table 6.2: Average expected runtime, average execution time and root mean square error for the first five datasets for training and testing on the same class. All runtimes are presented in milliseconds.

Measures	**Model**	**Amazon-GP**			**DBLP-ACM**			**DBLP-Scholar**		
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**	**expected**	**execution**	**RMSE**
PPJoin+	**exp**	7.33	14.45	2.78	8.36	14.56	2.43	177.02	124.88	8.02
	linear	8.37	16.24	3.28	7.45	15.81	2.97	222.55	147.33	9.48
	mixed	6.09	13.45	2.70	6.12	16.83	3.56	129.63	149.82	6.69
Ed-Join	**exp**	22.81	27.33	3.89	34.33	36.84	3.49	428.93	324.79	12.31
	linear	17.99	26.04	2.60	25.29	35.85	3.35	354.97	404.06	9.65
	mixed	18.34	26.45	2.78	27.68	41.20	3.54	338.55	339.31	7.30
Measures	**Model**	**MOVIES**			**VILLAGES**			**AVERAGE**		
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
PPJoin+	**exp**	134.90	146.39	5.44	211.89	135.53	9.36	*PPJoin+*	**exp**	5.61
	linear	38.60	33.10	2.95	158.89	131.64	5.23		**linear**	**4.78**
	mixed	48.45	49.89	3.17	214.15	201.17	8.13		**mixed**	4.85
Ed-Join	**exp**	59.57	45.47	3.76	1,225.57	1,556.04	35.23	*Ed-Join*	**exp**	11.74
	linear	43.02	44.46	3.52	509.71	294.35	22.53		**linear**	8.33
	mixed	45.55	43.26	2.88	377.02	286.91	10.89		**mixed**	**5.48**

To continue with Q_2, we conducted a set of experiments in order to observe how well each model could generalize among the different classes included in our evaluation data. Tables 6.3, 6.4, 6.5, 6.6 and 6.7 present the results of training on one dataset, and testing resulting models on the set of remaining classes. The highest RMSE error was achieved when both measures were tested using the *exp* model in all datasets except VILLAGES. However, Table 6.7 shows that the fitting error when trained on VILLAGES is relatively low in all three models. Additionally, we observe that the *exp* model's RMSE increased exponentially as the quantity of the training data decreased, which renders this model inadequate and unreliable for runtime approximations. By observing Tables 6.4 and 6.5, we see that the RMSE of the *exp* model increased by 38 orders of magnitude for *Ed-Join*.

For both measures, on average, the *linear* model outperformed the other two models when trained on the Amazon-GP, DBLP-ACM and DBLP-Scholar datasets, and achieved the lowest RMSE when trained on MOVIES for *Ed-Join*, in comparison to *exp* and *mixed*. Both *linear* and *mixed* models achieved minuscule approximation errors compared to *exp*, but *linear* was able to produce at least 35% less RMSE compared to *mixed*. Therefore, we can answer Q_2 by stating that the *linear* model is the most suitable and sufficient model that can generalize among different classes.

For our last question, we tested the performance of the different models when trained on a bigger and more diverse dataset. Table 6.8 shows the results of our evaluation, where each

Table 6.3: Average expected runtime, average execution time and root mean square error for training on the Amazon-GP dataset, and testing on DBLP-ACM, DBLP-Scholar, MOVIES and VILLAGES. All runtimes are presented in milliseconds.

Measures	Model	DBLP-ACM			DBLP-Scholar			AVERAGE		
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
PPJoin+	**exp**	18.24	64.02	8.61	1.84E+17	1,609.71	1.84E+16	*PPJoin+*	**exp**	8.42E+35
	linear	25.42	87.68	12.23	409.98	474.82	20.59		**linear**	**24.68**
	mixed	44.67	137.54	18.72	270.33	339.06	20.02		**mixed**	90.07
Ed-Join	**exp**	62.62	142.76	15.67	5.34E+19	834.11	5.34E+18			
	linear	37.19	131.68	19.26	663.07	837.88	27.30			
	mixed	38.36	140.25	16.87	770.51	861.72	21.91			
Measures	Model	MOVIES			VILLAGES					
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
PPJoin+	**exp**	8.79E+05	95.28	8.79E+04	3.37E+37	352.77	3.37E+36	*Ed-Join*	**exp**	8.43E+41
	linear	133.06	202.34	11.32	853.58	331.61	54.62		**linear**	**28.01**
	mixed	136.17	98.58	6.37	3,507.19	360.03	315.15		**mixed**	54.49
Ed-Join	**exp**	1.26E+07	143.93	1.26E+06	9.75E+42	6,108.37	9.75E+41			
	linear	209.13	142.45	9.14	1,379.12	864.31	56.32			
	mixed	332.13	145.46	19.83	7,258.82	5,973.70	159.37			

Table 6.4: Average expected runtime, average execution time and root mean square error for training on the DBLP-ACM dataset, and testing on Amazon-GP, DBLP-Scholar, MOVIES and VILLAGES. All runtimes are presented in milliseconds.

Measures	Model	Amazon-GP			DBLP-Scholar			AVERAGE		
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
PPJoin+	**exp**	21.51	61.69	9.93	1.29E+16	3,741.58	1.29E+15	*PPJoin+*	**exp**	3.99E+15
	linear	15.73	46.13	8.95	346.71	3,674.06	341.87		**linear**	**101.82**
	mixed	44.09	120.62	12.82	534.41	1,833.07	139.71		**mixed**	531.95
Ed-Join	**exp**	85.53	92.78	8.02	2.82E+18	888.50	2.82E+17			
	linear	56.95	90.10	7.91	950.61	883.01	25.97			
	mixed	58.29	96.63	8.48	1,472.52	881.22	63.72			
Measures	Model	MOVIES			VILLAGES					
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
PPJoin+	**exp**	8.05E+05	108.16	8.05E+04	1.47E+37	356.93	1.47E+36	*Ed-Join*	**exp**	9.3E+42
	linear	127.07	132.62	7.64	819.98	368.86	48.82		**linear**	**53.95**
	mixed	159.36	120.74	8.92	2.14E+04	1,783.72	1,966.38		**mixed**	1,105.15
Ed-Join	**exp**	3.58E+07	156.97	3.58E+06	3.72E+44	6,329.54	3.72E+43			
	linear	373.99	156.72	23.23	2,440.64	870.15	158.72			
	mixed	1,246.20	155.42	109.39	4.87E+04	6,411.76	4,239.01			

Table 6.5: Average expected runtime, average execution time and root mean square error for training on the DBLP-Scholar dataset, and testing on Amazon-GP, DBLP-ACM, MOVIES and VILLAGES. All runtimes are presented in milliseconds.

Measures	Model	Amazon-GP			DBLP-ACM			AVERAGE		
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
PPJoin+	**exp**	79.32	65.28	8.03	47.42	69.70	8.74	*PPJoin+*	**exp**	4.56E+04
	linear	-364.95	38.47	40.61	173.40	88.48	15.39		**linear**	**85.07**
	mixed	-41.05	50.27	11.00	-148.99	88.14	26.03		**mixed**	427.54
Ed-Join	**exp**	113.56	80.90	8.67	113.43	139.78	16.74			
	linear	44.49	79.97	10.67	37.70	144.33	22.36			
	mixed	40.13	73.76	8.98	40.94	141.33	18.84			
Measures	Model	MOVIES			VILLAGES					
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
PPJoin+	**exp**	110.41	94.69	6.31	1.82E+06	1,546.07	1.82E+05	*Ed-Join*	**exp**	1.10E+04
	linear	394.74	104.19	29.99	3,158.25	621.84	254.30		**linear**	**54.57**
	mixed	66.96	85.61	6.76	1.82E+04	1,591.24	1,666.38		**mixed**	82.52
Ed-Join	**exp**	341.02	128.33	22.66	4.46E+05	6,069.92	4.41E+04			
	linear	360.47	127.76	24.51	2,418.34	818.14	160.73			
	mixed	280.77	125.19	16.86	3,670.31	820.85	285.43			

model was trained on DBpedia english labels and tested on the four evaluation datasets. The *linear* model error was 1 order of magnitude less than the RMSE obtained by *exp* and 3 orders

Table 6.6: Average expected runtime, average execution time and root mean square error for training on the MOVIES dataset, and testing on Amazon-GP, DBLP-ACM, DBLP-Scholar and VILLAGES. All runtimes are presented in milliseconds.

Measures	Model	Amazon-GP			DBLP-ACM			AVERAGE		
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
	exp	19.53	71.55	7.89	46.89	127.70	15.90			
PPJoin+	**linear**	-45.99	42.58	10.51	57.73	120.70	23.93		**exp**	8.42E+06
	mixed	16.97	39.64	5.84	17.43	66.84	9.77	*PPJoin+*	**linear**	51.34
	exp	15.57	80.95	9.37	16.24	135.66	17.93		**mixed**	**37.99**
Ed-Join	**linear**	1.71	84.53	10.82	3.56	138.18	19.89			
	mixed	4.33	85.70	10.95	6.99	140.99	19.65			
Measures	**Model**	**DBLP-Scholar**			**VILLAGES**					
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
	exp	3,636.56	318.89	332.11	3.37E+08	634.17	3.37E+07			
PPJoin+	**linear**	372.82	1,315.61	102.21	1,064.96	389.93	68.69		**exp**	1.46E+06
	mixed	75.49	702.11	67.82	989.17	311.60	68.54	*Ed-Join*	**linear**	**25.91**
	exp	4,060.80	811.77	325.48	5.85E+07	767.66	5.85E+06		**mixed**	42.85
Ed-Join	**linear**	259.61	805.29	57.92	696.29	753.35	15.04			
	mixed	178.93	796.16	65.09	1,522.63	777 .00	75.74			

Table 6.7: Average expected runtime, average execution time and root mean square error for training on the VILLAGES dataset, and testing on Amazon-GP, DBLP-ACM, DBLP-Scholar and MOVIES. All runtimes are presented in milliseconds.

Measures	Model	Amazon-GP			DBLP-ACM			AVERAGE		
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
	exp	93.41	67.44	5.08	35.07	62.53	8.36			
PPJoin+	**linear**	-192.27	24.57	21.87	-133.03	61.10	21.09		**exp**	**10.16**
	mixed	16.37	32.66	3.40	41.57	61.83	9.20	*PPJoin+*	**linear**	22.91
	exp	68.00	53.36	4.50	326.05	143.84	26.53		**mixed**	30.59
Ed-Join	**linear**	-123.44	55.03	18.20	-677.4	133.63	82.36			
	mixed	231.61	50.46	18.51	136.49	139.30	15.95			
Measures	**Model**	**DBLP-Scholar**			**MOVIES**					
		expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**			
	exp	92.10	272.40	21.78	56.74	82.92	5.43			
PPJoin+	**linear**	-39.98	277.56	34.10	-54.33	84.08	14.57		**exp**	**21.59**
	mixed	84.22	451.80	40.04	-26.91	651.50	69.71	*Ed-Join*	**linear**	54.56
	exp	316.66	784.70	49.85	138.63	114.50	5.46		**mixed**	32.75
Ed-Join	**linear**	159.75	753.00	61.23	-438.84	122.89	56.44			
	mixed	1,737.75	945.09	81.94	255.96	116.42	14.61			

of magnitude less than the *mixed* error. In all four datasets, the *mixed* model produced the highest RMSE. For the VILLAGES dataset, the *mixed* model's error was 1,916 and 214 times higher than *linear* and *exp* resp. Figures. 6.2 and 6.3 present the plans produced by Helios for the LS illustrated in Figure 6.1 of the Amazon-GP dataset, if the planner used the *exp* model and the *linear* or the *mixed* models resp. For the left sub-specification

```
AND( levenshtein (x. description ,y. description )|0.5045 ,
 trigrams(x. title , y.name)|0.4871) >= 0.2925
```

the *linear* and the *mixed* models chose to execute only `trigrams(x.title, y.name)|0.4871` and use the other sub-specification as filters. Moreover, the plan retrieved by using the *exp* model for runtime approximations aims to execute both children LSs, which results in an overhead in the execution of the LS. It is obvious that, on average, the *linear* model achieved by far the lowest RMSE compared to the other two models, which concludes the answer to Q_3.

Table 6.8: Average expected runtime, average execution time and root mean square error for training on DBPedia english labels and testing on Amazon-GP, DBLP-ACM, MOVIES and VILLAGES. All runtimes are presented in milliseconds.

Model			**Amazon-GP**			**DBLP-ACM**		**AVERAGE**
	expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**		
exp	5,242.09	3,618.99	3,164.86	308.14	365.46	126.42		
linear	300.51	3,043.97	966.99	8.07	361.53	192.12		
mixed	-7.27E+06	4,512.82	6.78E+05	-7.26E+04	310.49	4.38E+04	**exp**	4,577.58
Model			**Amazon-GP**			**DBLP-ACM**	**linear**	**512.35**
expected	**execution**	**RMSE**	**expected**	**execution**	**RMSE**		**mixed**	9.82E+05
exp	584.27	1,061.67	160.05	4.61E+04	3,775.54	1.48E+04		
linear	323.04	995.04	258.55	2,626.41	3,832.52	631.72		
mixed	-3,417.80	1,600.81	2,042.45	7.15E+06	3,891.05	3.20E+06		

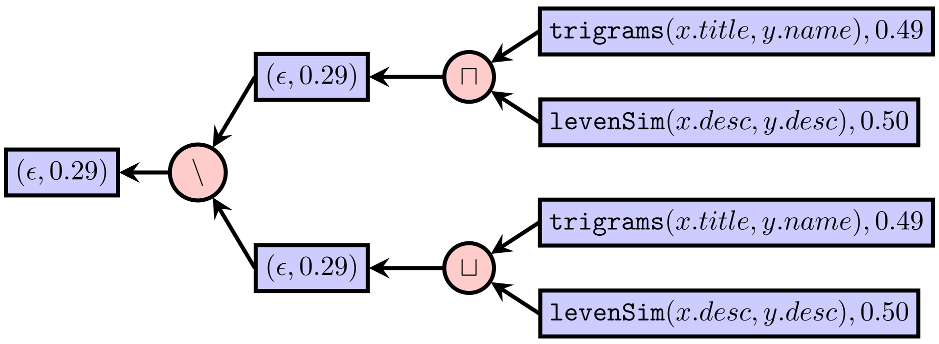

trigrams($x.title, y.name$), 0.49
$(\epsilon, 0.29)$
$\sqcap$
levenSim($x.desc, y.desc$), 0.50
$(\epsilon, 0.29)$
$\setminus$
trigrams($x.title, y.name$), 0.49
$(\epsilon, 0.29)$
$\sqcup$
levenSim($x.desc, y.desc$), 0.50

Figure 6.1: Example LS from the Amazon-GP dataset. Note that we used `desc` for *description* and `levenSim` for the Levenshtein similarity.

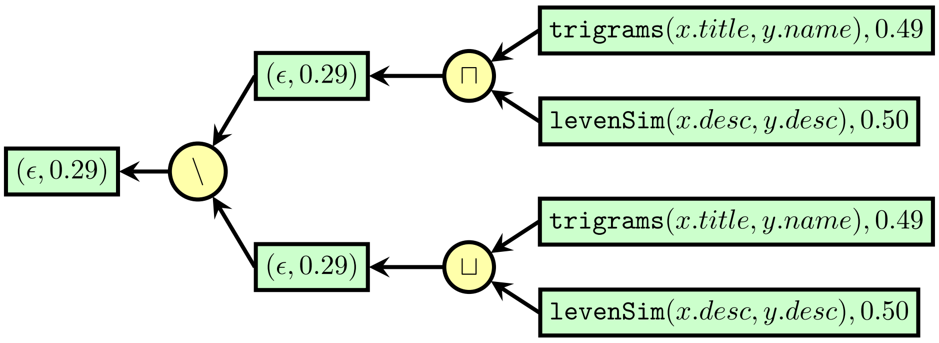

trigrams($x.title, y.name$), 0.49
$(\epsilon, 0.29)$
$\sqcap$
levenSim($x.desc, y.desc$), 0.50
$(\epsilon, 0.29)$
$\setminus$
trigrams($x.title, y.name$), 0.49
$(\epsilon, 0.29)$
$\sqcup$
levenSim($x.desc, y.desc$), 0.50

Figure 6.2: Plan returned from Helios using the *exp* model. Note that we used `desc` for *description* and `levenSim` for the Levenshtein similarity.

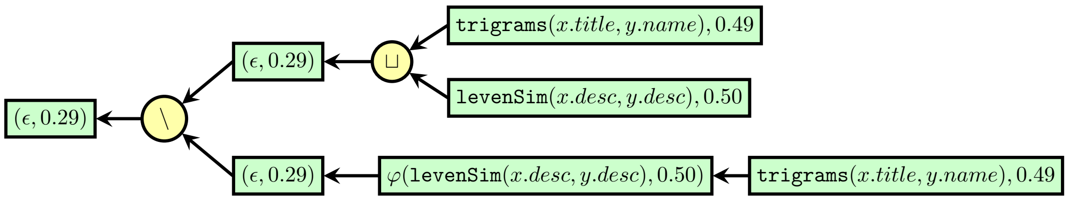

trigrams($x.title, y.name$), 0.49
$(\epsilon, 0.29)$
$\sqcup$
levenSim($x.desc, y.desc$), 0.50
$(\epsilon, 0.29)$
$\setminus$
$(\epsilon, 0.29)$
φ(levenSim($x.desc, y.desc$), 0.50)
trigrams($x.title, y.name$), 0.49

Figure 6.3: Plan returned from Helios using the *linear* and *mixed* model. Note that we used `desc` for *description* and `levenSim` for the Levenshtein similarity.

7 Liger: Link Discovery with Partial Recall

Preamble This chapter is based on [69]. It is the first approach that addresses the problem of partial-recall for LD. The author co-designed, implemented and evaluated the algorithm presented herein, and co-wrote the paper.

7.1 Linking with Guaranteed Expected Recall

In this section, we present our approach to achieve a guaranteed expected recall when provided with an input LS L. We begin by giving a formal definition of the selectivity function and partial-recall Link Discovery. We continue by presenting observations related to a quasi-ordering for a LS. Thereafter, we present an operator for link specifications, which allows Link Discovery with guaranteed partial selectivity, and prove some of its theoretical characteristics. We later use these characteristics to provide an efficient implementation of our operator.

7.1.1 Partial-Recall Link Discovery

A selectivity function $sel : \mathcal{LS} \rightarrow [0,1]$, where $\mathcal{LS}$ is the set of all LSs, encodes the predicted value of $|[[L]]|$ as a fraction of $|S \times T|$. This is akin to the selectivity definition often used in the database literature. A specification L' is said to achieve a recall k w.r.t. to L if $k \times |[[L]]| = |[[L]] \cap [[L']]|$. If $[[L']] \subseteq [[L]]$, then the recall k of L' abides by the simpler equation $k \times |[[L]]| = |[[L']]|$. A specification $[[L']] \subseteq [[L]]$ is said to achieve an expected recall k w.r.t. to L if $k \times sel(L) = sel(L')$.

Definition 7.1 (Partial-Recall Link Discovery)**.** *Given a specification L, the aim of partial-recall LD is to detect a rapidly executable LS $L' \sqsubseteq L$ with an expected recall of at least $k \in [0,1]$, i.e. a LS L' with $sel([[L']]) \geq k \times sel([[L]])$, where $k \in [0,1]$ is a minimal expected recall requirement set by the user.*

7.1.2 Subsumption of Link Specifications

Definition 7.2 (Subsumption of Link Specifications)**.** *The LS L is subsumed by the LS L' (denoted $L \sqsubseteq L'$) when $[[L]] \subseteq [[L']]$ for all fixed pair of sets S and T.*

Note that $\sqsubseteq$ is a quasi-ordering (i.e., reflexive and transitive) on $\mathcal{LS}$. A key observation that underlies our approach is the following:

Proposition 1. $\forall \theta, \theta' \in [0,1]\ \theta > \theta' \rightarrow (m(p_s,p_t),\theta) \sqsubseteq (m(p_s,p_t),\theta')$.

Proof. This is due to the definition of the subsumption relation as $m(p_s, p_t) \geq \theta \wedge \theta > \theta'$ implies that $m(p_s, p_t) \geq \theta'$ by virtue of the strict ordering on numbers in $\mathbb{R}$. $\square$

This observation can be extended to specifications as follows:

1. $L_1 \sqsubseteq L'_1 \rightarrow (L_1 \sqcup L_2) \sqsubseteq (L'_1 \sqcup L_2)$
2. $L_1 \sqsubseteq L'_1 \rightarrow (L_1 \sqcap L_2) \sqsubseteq (L'_1 \sqcap L_2)$
3. $L_1 \sqsubseteq L'_1 \rightarrow (L_1 \backslash L_2) \sqsubseteq (L'_1 \backslash L_2)$
4. $L_2 \sqsubseteq L'_2 \rightarrow (L_1 \backslash L'_2) \sqsubseteq (L_1 \backslash L_2)$

Proposition 2. $\sqsubseteq$ *is a quasi-ordering (i.e., reflexive and transitive) on* $\mathcal{LS}$.

Proof. $\sqsubseteq$ being a quasi-ordering is a direct consequence of the reflexivity and transitivity of $\subseteq$. $\square$

7.1.3 Refinement of a Link Specification for Guaranteed Selectivity

Definition 7.3 (Refinement Operator)**.** *In the quasi-ordered space* $(\mathcal{LS}, \sqsubseteq)$*, we call any function* h*:* $\mathcal{L} \rightarrow 2^{\mathcal{LS}}$ *an (LS)* operator*. A* downward refinement operator ρ *is an operator such that for all* $L \in \mathcal{LS}$ *we have* $L' \in \rho(L)$ *implies* $L' \sqsubseteq L$*.* L' *is called a* specialisation *of* L*. We denote* $L' \in \rho(L)$ *with* $L \rightsquigarrow_\rho L'$*.*

The idea behind our approach is to use a refinement operator to transform the input LS L into a LS $L' \sqsubseteq L$ with at least a given expected recall k. We define the corresponding refinement operator over the space $(2^{\mathcal{LS}}, \sqsubseteq)$ as follows:

$$\rho(L) = \begin{cases} \emptyset & \text{if } L = L_\emptyset, \\ L_\emptyset & \text{if } L = (m(p_s, p_t), 1), \\ (m(p_s, p_t), next(\theta)) & \text{if } L = (m(p_s, p_t), \theta) \wedge \theta < 1, \\ (\rho(L_1) \sqcup L_2) \cup (L_1 \sqcup \rho(L_2)) & \text{if } L = L_1 \sqcup L_2, \\ (\rho(L_1) \sqcap L_2) \cup (L_1 \sqcap \rho(L_2)) & \text{if } L = L_1 \sqcap L_2, \\ \rho(L_1) \backslash L_2 & \text{if } L = L_1 \backslash L_2. \end{cases} \tag{7.1}$$

This operator works as follows:

- If L is the empty specification $L_\emptyset$, then we return an empty set of specifications, ergo, L is not refined any further.
- *If L is an atomic specification* with a threshold of 1, our approach returns $L_\emptyset$. By these means, we can compute refinement chains from $L = L_1 \sqcup L_2$ to $\{L_1, L_2\}$. If $\theta < 1$, our approach alters the threshold θ by applying the *next* function. This function is based on the insight that for $\theta < 1$ and any pair of datasets S and T, if there is the smallest threshold $\theta' > \theta$ that leads to $[[(m(p_s, p_t), \theta')]] \subset [[(m(p_s, p_t), \theta)]]$, then θ' is a finite non-zero positive real number. This is exactly the value that $next(\theta)$ returns for each metric if θ' exists. If θ' does not exist, $[[(m(p_s, p_t), \theta)]]$ is the same as $[[(m(p_s, p_t), 1)]]$. Then, *next* returns $\varnothing$, which is evaluated like $L_\emptyset$. Formally, for a given set of input datasets S and T and any θ, *next* always returns values from $N(m(p_s, p_t)) = \{n : \exists (s, t) \in S \times T : n = m(p_s, p_t)\} \cup \{\varnothing\}$. Given a threshold θ, $\varnothing$ is returned if L is to be refined to the empty specification. Otherwise, *next* returns the smallest value from $N(m(p_s, p_t))$ that is larger than θ. Note that $(m(p_s, p_t), next(\theta)) \sqsubseteq (m(p_s, p_t), \theta)$ always holds. For example, for the right sub-specification of the LS shown in Figure 7.1, $next(0.7)$ would return 1.

- If L *is complex*, then the refinement depends on the operator of the specification and always ensures that $L' \in \rho(L) \rightarrow L' \sqsubseteq L$. To explicate the set of LSs returned by ρ, we extend the semantics of $op(\rho(L_1), L_2)$ resp. $op(L_1, \rho(L_2))$ to be the set of all specifications that can be computed by using op on all $L' \in \rho(L_1)$ resp. $L' \in \rho(L_2)$.
  - If $op = \sqcap$ or $op = \sqcup$, then ρ returns the union of all specifications that can be generated by applying ρ to one sub-specification of L and combining these with the other sub-specification.
  - If $op = \backslash$, then we combine L's right sub-specification with all refinements of L's left sub-specification. The reason we do not do this the other way around for this particular operator is simply that ρ would not be a refinement operator if we did so, as we could not guarantee that $\rho(L) \sqsubseteq L$.

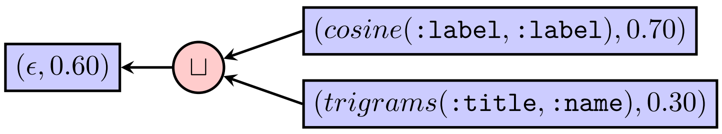

$(\epsilon, 0.60)$
$\sqcup$
$(cosine(\texttt{:label}, \texttt{:label}), 0.70)$
$(trigrams(\texttt{:title}, \texttt{:name}), 0.30)$

Figure 7.1: Tree representation of a complex LS

A refinement operator ϱ over the quasi-ordered space $(\mathcal{S}, \preccurlyeq)$ can abide by the following criteria.

Definition 7.4 (Finiteness)**.** *ϱ is finite iff $\varrho(s)$ is finite for all $s \in \mathcal{S}$.*

Definition 7.5 (Properness)**.** *ϱ is proper if $\forall s \in \mathcal{S}, s' \in \varrho(s) \Rightarrow s \neq s'$.*

Definition 7.6 (Completeness)**.** *ϱ is said to be complete if for all s and s', $s' \preccurlyeq s$ implies that there is a s'' with $s'' \preccurlyeq s' \wedge s' \preccurlyeq s''$ such that a refinement chain between s'' and s exists.*

Definition 7.7 (Redundancy)**.** *A refinement operator ϱ over the space $(\mathcal{S}, \preccurlyeq)$ is redundant if two different refinement chains can exist between $s \in \mathcal{S}$ and $s' \in \mathcal{S}$.*

Having defined our refinement operator, we now show that ρ is finite, incomplete, proper and redundant if L, S and T are finite.

Proposition 3. *ρ is finite if the input specification is finite.*

Proof. This is a direct consequence of $next(\theta)$ being finite. The number of elements in $\rho(L)$ is at most equal to the $|L|$. Now if $|L|$ is finite as required in the premise of this proof, then $|\rho(L)|$ must also be finite. □

Proposition 4. *ρ is proper.*

Proof. This is a direct consequence of $next(\theta)$ returning a θ' which alters the result of the specification to which it is applied and is thus different from θ. Hence, $\rho(L) \neq L$, which implies that ρ is proper. □

Proposition 5. *ρ is incomplete.*

Proof. Given any specification L, ρ cannot generate $L \sqcap L'$ although $(L \sqcap L') \sqsubseteq L$ clearly holds. □

This is not a restriction for our purposes given that we aim to find specifications that run faster. Thus we do not want to extend the input specification L by combining it with other specifications L' that might make our implementation of the operator slower. Note that our operator can simplify specifications by removing $\sqcup$ by virtue of $\varnothing$ in the definition of the *next* function. Moreover, given a specification L, our operator can generate all specifications that differ from L solely w.r.t. the thresholds in their leaves and lead to different mappings. This is simply due to our definition of the *next* function.

Proposition 6. *ρ is redundant.*

Proof. Given any specification $L = (m_1(p_s, p_t), \theta_1) \sqcap (m_2(p_s, p_t), \theta_2)$, there are two refinement chains to $L' = (m_1(p_s, p_t), next(\theta_1)) \sqcap (m_2(p_s, p_t), next(\theta_2))$. These are:

1. $L \rightsquigarrow_\rho (m_1(p_s, p_t), \theta_1) \sqcap (m_2(p_s, p_t), next(\theta_2)) \rightsquigarrow_\rho L'$ and
2. $L \rightsquigarrow_\rho (m_1(p_s, p_t), next(\theta_1)) \sqcap (m_2(p_s, p_t), \theta_2) \rightsquigarrow_\rho L'$.

□

The meaning of the other characteristics for our implementation of ρ is as follows: given that ρ is *finite*, we can completely generate ρ for any chosen node in our implementation. *ρ being redundant* means that after a refinement, we need to check whether we have already seen the newly generated specifications. Hence, we need to keep a set of seen specifications. Finally, the *operator being proper* means that while checking for redundancy, there is no need to compare specifications with any of their parents.

7.2 The Liger approach

Let L_0 be a LS and $\rho^*(L_0)$ represent the set of all LSs that can be reached from L_0 via ρ. The basic goal behind Liger (LInk discovery with Guaranteed Expected Recall) is to find the LS $\Lambda \in \rho^*(L_0)$ that achieves the lowest expected run time while (1) being subsumed by L_0 and (2) achieving at least a predefined expected recall $k \in [0, 1]$.

We denote this selectivity constraint with $k \in [0, 1]$, which is formally the minimal fraction of the selectivity of L_0 that Λ must achieve. To achieve this goal, we use the downward refinement operator ρ described in Section 7.1.3 to search through the space of $\mathcal{LS}$ subsumed by L_0 while only refining LSs that abide by the conditions above.

In simple terms, for any LS L, Liger assumes that it can (1) approximate its run time using a linear model described in Chapter 6, and (2) estimate its selectivity as follows:

- For an atomic LS, the selectivity values were computed using $\frac{|[[L]]|}{|S| \times |T|}$, where $|[[L]]|$ is the size of the mapping returned by the LS L, $|S|$ and $|T|$ are the sizes of the source and target data. To do so, we pre-computed the real selectivity of atomic LSs that were based on a set of measures [1] using the methodology presented in [138] for thresholds between 0.1 and 1.
- For complex LSs, which are binary combinations of two LSs L_1 (selectivity: $sel(L_1)$) and L_2 (selectivity: $sel(L_2)$), the run time approximation was computed by summing up the individual run times of L_1,L_2, in addition to a constant value of 1 for each operator. The selectivity of operators was computed based on the selectivity of the mappings that served

[1](1) `cosine`, `levenshtein`, `qgrams`, `trigrams`, `jaccard` and `overlap` for string properties, (2) `euclidean` for numeric properties and (3) `geomean` and `hausdorff` for geometric properties in the form of point sets

as input for the operators. We assumed the selectivity of a LS L to be the probability that a pair (s, t) is returned after applying L. Based on these assumptions, we derived the following selectivities:

- $op(L) = \cap \rightarrow sel(L) = \frac{1}{2}sel(L_1)sel(L_2)$,
- $op(L) = \cup \rightarrow sel(L) = \frac{1}{2}(1 - (1 - sel(L_1))(1 - sel(L_2)))$ and,
- $op(L) = \setminus \rightarrow sel(L) = \frac{1}{2}sel(L_1)(1 - sel(L_2))$.

Note that we used a correction factor of 0.5 to correct for the dependence of the selectivity across properties like in previous works [2].

The approach starts by initializing a refinement tree with the given LS L_0. Then, it selects the previously unvisited LS of the tree that:

1. has the lowest expected run time, and
2. abides by the settings provided by the user.

LIGER then computes the complete refinement of the selected LS by virtue of ρ's finiteness. Redundant refinement results are subsequently detected (as ρ is redundant) and not added into the tree. These steps are carried out until the refinement tree has been explored completely or a time threshold for running the refinements is met. In the following, we present the approach in detail as well as a variation on the approach that makes use of the potential monotonicity of specification run times.

7.2.1 The LIGER Algorithm

Algorithm 12 shows the steps of the basic LIGER implementation. We dub this implementation C-RO (ReCall with Refinement Operator). Our approach takes (1) a LS L_0, (2) one input source KB S and one input target KB T, (3) the minimal expected recall k and (4) a refinement time constraint *maxOpt* as input. We begin by retrieving the estimations of the selectivity of L_0 for the given S and T (sel_{L_0}, see line 1 of Algorithm 12). The algorithm computes the desired selectivity value (sel_{des}) as a fraction of L_0's selectivity (line 2). We then initialize the best subsumed LS Λ with L_0 and the best run time rt_Λ with L_0's runtime estimation for a particular S and T (line 3, line 4). Then, we add L_0 to the set $Buffer$ (line 5). This set serves as a buffer and includes LSs obtained by refining L_0 that have not yet been refined. All LSs generated through the refinement procedure, as well as the input LS L_0 are stored in $Total$ (line 6). By keeping track of these LSs, we avoid refining a LS more than once and address the redundancy of our refinement operator.

The main loop starts in line 7 and runs until the termination criterion is met, i.e., until the refinement time has exceeded *maxOpt*, the refinement tree cannot be explored further ($Buffer = \emptyset$) or the selectivity of Λ is equal to sel_{des}. We define the refinement tree as follows: (1) it has L_0 as its root, (2) at each iteration of Algorithm 12, the set of refined LSs are added as children nodes to the currently refined LS, and (3) a leaf node is as a LS that cannot be refined any further. At the beginning of each iteration, the algorithm selects the next node for refinement by calling the function $getNextNode(Buffer, S, T)$(line 8). $getNextNode(Buffer, S, T)$ computes a complete ordering of $Buffer$ with respect to runtime estimations. Algorithm 13 illustrates the steps of the algorithm. Initially, the $getNextNode(Buffer, S, T)$ algorithm retrieves the run time estimation (rt_L) of each LS $L \in Buffer$ (line 3) and adds the pair (L, rt_L) to the set $Nodes$. Then, if $Nodes$ is not empty, it orders the LS of $Nodes$ based on the run time scores in ascending order and sets to L_{next} the first element of the set (line 7, line 8 resp. of Algorithm 13). If $Nodes$ is empty, then Algorithm 12 terminates (line 10 of Algorithm 12).

Next, Algorithm 12 checks if the current LS L receives a better run time score, the algorithm assigns L as the new value of Λ and changes rt_Λ accordingly (line 14).

Algorithm 12: Liger Algorithm

```
Input: a link specification $L_0$; two input KBs $S$ and $T$;
minimal expected recall $k$
Output: subsumed link specification $\Lambda$
1  $sel_{L_0} \leftarrow getSelectivity(L_0, S, T)$
2  $sel_{des} \leftarrow sel_{L_0} * k$
3  $\Lambda \leftarrow L_0$
4  $rt_\Lambda \leftarrow getRuntime(L_0, S, T)$
5  $Buffer \leftarrow \{L_0\}$
6  $Total \leftarrow \{L_0\}$
7  while termination criterion not met do
8      $L \leftarrow getNextNode(Buffer, S, T)$
9      if $L == null$ then
10         break;
11     $rt_L \leftarrow getRuntime(L, S, T)$
12     if $rt_L < rt_\Lambda$ then
13         $\Lambda \leftarrow L$
14         $rt_\Lambda \leftarrow rt_L$
15     $newLSs \leftarrow refine(L)$
16     if $newLSs \neq \emptyset$ then
17         $update(newLSs, Total, Buffer, sel_{des}, S, T)$
18     $Buffer \leftarrow Buffer \setminus L$
19 Return $\Lambda$
```

In line 15, the algorithm calls the function $refine(L)$, which implements ρ. The results of the refinement are stored in $newLSs$. Algorithm 14 illustrates the steps towards the refinement of a LS L. First, it checks whether L is atomic and if its threshold θ is less than 1 (line 2). If the condition is fulfilled, the algorithm calls the $next(\theta)$ function (line 3) to retrieve the new threshold θ', then it creates a new LS L' that has the same similarity function as L and θ' as threshold (line 4). Finally, the L' is added to the set $newLSs$ and returned to the main Liger algorithm. If L is not atomic, Algorithm 14 calls the function $refine(L)$ recursively for the left child of L in line 7. If the returned set of refined LSs obtained from the left child is not empty, the algorithm proceeds in merging all LSs of L_{ref1} with the right child, which was not refined based on Equation 7.1 (line 10). The new LSs obtained from the merge are then added to $newLSs$ in line 11. Once the left child of L has been refined, the algorithm checks if $op(L)$ is equal to $\setminus$. If the condition does not hold, the algorithm performs the same procedure of refining the right child of L as it did with the left child (lines 13- 17).

Once $newLSs$ has been retrieved (line 15), Algorithm 12 must check which subsumed LS(s) of $newLSs$ can be refined in the future. To this end, it calls the function $update(newLSs, Total, Buffer, sel_{des}, S, T)$ (line 17 of Algorithm 12). This methods checks that each LS $L' \in newLSs$ has not been explored before by checking if it already exists in set $Total$ (line 2). Therewith, we ensure that Liger does not explore LSs that have already been seen before. If L' is a new node, it is added to $Total$ (line 3) and the algorithm proceeds in computing L''s selectivity (line 4). If $sel_{L'}$ is higher or equal to the desired selectivity, the algorithm updates $Buffer$ by adding

Algorithm 13: $getNextNode(Buffer, S, T)$ for C-RO

```
Input: set of unrefined nodes $Buffer$; two input KBs $S$ and $T$
Output: a node for refinement $L_{next}$
1 $Nodes \leftarrow \emptyset$
2 $L_{next} \leftarrow null$
3 foreach $L \in Buffer$ do
4     $rt_L \leftarrow getRuntime(L, S, T)$
5     $Nodes \leftarrow Nodes \cup (L, rt_L)$
6 if $Nodes \neq \emptyset$ then
7     $Nodes \leftarrow order(Nodes)$
8     $L_{next} \leftarrow Nodes.getTop()$
9 Return $L_{next}$
```

L' (line 6)[2]. Finally, in Algorithm 12, after L has been refined, it is excluded from $Buffer$ (line 18), so it will not be refined in the future.

7.2.2 Extension of Liger

One key observation pertaining to the run time of $L' \in \rho^*(L)$ is that by virtue of $L' \sqsubseteq L$, $rt_{L'} \leq rt_L$ will most probably hold. By virtue of the transitivity of $\leq$, $L_1 \in \rho(L) \wedge L_2 \in \rho(L) \wedge rt_{L_1} \leq rt_{L_2} \rightarrow \forall L' \in \rho^*(L_1)$: $rt_{L'} \leq rt_{L_2}$ also holds. We call this assumption the *monotonicity of run times.* Since the implementation of Algorithm 13 for Liger does not take this monotonicity into consideration, we wanted to know whether this assumption can potentially improve the run time of our approach. A direct consequence of this assumption would be the following:

Proposition 7. *Let L_1 be a leaf of the refinement tree at the distance l from the root of ρ's refinement tree, then for all leaves L_2 at a distance $l' > l$ from the root, $rt_{L_1} \geq rt_{L_2}$.*

Proof. $rt_{L_1} \geq rt_{L_2}$ is a direct consequence of the approach behind Liger. If the distance l' of L_2 from the root of the refinement tree is larger than l, then L_2 must have an ancestor L_2' in the refinement tree at the distance l of the root. Now if L_1 is a leaf, then L_2' was preferred over L_1 when refining, hence $rt_{L_1} \geq rt_{L_{2'}}$. By virtue of the monotonicity, $rt_{L_{2'}} \geq rt_{L_2}$, hence $t_{L_1} \geq t_{L_2}$ □

To integrate it into Liger, we created the extension of Liger dubbed RO-MA (<u>R</u>efinement <u>O</u>perator with <u>M</u>onotonicity <u>A</u>ssumption) (Refinement Operator with Monotonicity Assumption). RO-MA overwrites the $getNextNode$ $(Buffer, S, T)$ function with $getNextNode(Total$, $Buffer, S, T)$ (line 8 of Algorithm 12) by using a hierarchical ordering on the set of unrefined nodes. By incorporating RO-MA as a search strategy, the refinement tree is expanded using a "top-down" approach until there are no nodes to be further explored in a particular path. Algorithm 16 describes the procedure used to find the next node for refinement. Initially, the algorithm retrieves the level (i.e., the distance from the root of the refinement tree, denoted lvl_{L_r}) of the most recently defined LS L_r and increases it by 1. The first time Algorithm 16 is called, the only node included in $Total$ will be L_0 with $lvl_{L_r} = 0$, since it is the root of the refinement tree. In this case, the algorithm will return L_0 to be refined (line 3). The main loop begins in line 8 and continues until lvl is equal to 0 (i.e., until the level of the root). Then,

[2] $Total$ is passed by reference and updated within the *update* function

Algorithm 14: $refine(L)$

```
Input: the LS $L$ that will be refined
Output: a set of new LSs $newLSs$ obtained by refining $L$
1  $newLSs \leftarrow \emptyset$
2  if $L = (m(p_s, p_t), \theta)$ and $\theta < 1.0$ then
3      $\theta' \leftarrow next(\theta)$
4      $L' \leftarrow (m(p_s, p_t), \theta')$
5      $newLSs \leftarrow newLSs \cup L'$
6  else
7      $L_{ref1} \leftarrow refine(L.leftChild)$
8      $L_2 \leftarrow L.rightChild$
9      if $L_{ref1} \neq \emptyset$ then
10         $tempD_1 \leftarrow merge(L_{ref1}, L_2)$
11         $newLSs \leftarrow newLSs \cup tempD_1$
12     if $L.operator \neq \setminus$ then
13         $L_1 \leftarrow L.leftChild$
14         $L_{ref2} \leftarrow refine(L.rightChild)$
15         if $L_{ref2} \neq \emptyset$ then
16             $tempD_2 \leftarrow merge(L_{ref2}, L_1)$
17             $newLSs \leftarrow newLSs \cup tempD_2$
18 Return $newLSs$
```

we find the set of nodes at level *lvl*, that have not be refined before (line 11). If such a subset exists (line 13), then the algorithm retrieves the run time estimation of each LS $L \in SubTree$, and adds it to the set *Nodes*. Then, it orders the LSs of *Nodes* based on the run time scores in ascending order (line 18) and sets to L_{next} the first element of the set (line 19). If all the nodes of *lvl* have been refined before, then the search for the next LS continues at a level higher. If the root's level has been reached, then Algorithm 12 terminates.

7.2.3 Example Run

To elucidate the workings of C-RO and RO-MA further, we use the LS described in Figure 7.1 as a running example. Table 7.1 shows the *estimated* runtime (*rt*) and selectivity (*sel*) for each subsumed LS. For our example, each call of the $next(\theta)$ function will return the θ increased by

Algorithm 15: $update(newLSs, Total, Buffer, sel_{des}, S, T)$

```
Input: set of new nodes $newLSs$, set of unrefined nodes $Buffer$; set of all nodes
       $Total$; the desired selectivity $sel_{des}$; two input KBs $S$ and $T$;
1  foreach $L' \in newLSs$ do
2      if $L' \notin Total$ then
3          $Total \leftarrow Total \cup L'$
4          $sel_{L'} \leftarrow getSelectivity(L', S, T)$
5          if $sel_{L'} \geq sel_{des}$ then
6              $Buffer \leftarrow Buffer \cup L'$
```

Algorithm 16: $getNextNode(Total,\ Buffer, S, T)$ for RO-MA

```
   Input: set of all nodes $Total$; set of unrefined nodes $Buffer$; two input KBs $S$ and $T$
   Output: a node for refinement $L_{next}$
1  $L_r \leftarrow Total.getLatestNode()$
2  $lvl_{L_r} \leftarrow Total.getLevel(L_r)$
3  if $lvl_{L_r} == 0$ then
4      $L_{next} \leftarrow Buffer.getTop()$
5      Return $L_{next}$
6  $lvl \leftarrow lvl_{L_r} + 1$
7  $L_{next} \leftarrow null$
8  while $lvl \neq 0$ do
9      $SubTree \leftarrow \emptyset$
10     foreach $L \in Buffer$ do
11         if $lvl_L == lvl$ then
12             $SubTree \cup L$
13     if $SubTree \neq \emptyset$ then
14         $Nodes \leftarrow \emptyset$
15         foreach $L \in SubTree$ do
16             $rt_L \leftarrow getRuntime(L, S, T)$
17             $Nodes \leftarrow Nodes \cup (L, rt_L)$
18         $Nodes \leftarrow order(Nodes)$
19         $L_{next} \leftarrow Nodes.getTop()$
20         break;
21     else $lvl \leftarrow lvl - 1$ ;
22 Return $L_{next}$
```

0.1. The k is set to 0.6 and $maxOpt = 10s$.

The Liger algorithm begins by assigning the LS of Figure 7.1 to Λ, and then adding it to both $Buffer$ and $Total$ sets (lines 5 and 6). Then, we set the values for $rt_\Lambda = 50$ and $sel_{des} = 0.57$. After that, the main loop begins, and for both C-RO and RO-MA, the $getNextNode(Buffer, S, T)$ or $getNextNode(Total, Buffer, S, T)$ resp. function will return the initial LS (⓪ from Table 7.1). Since ⓪ is not $null$ and its estimated runtime is not less than the rt_Λ, the main algorithm will invoke the $refine(L)$ function that will fill the set $newLSs$ with ① and ②. Since both LSs are new, they are going to be added to the $Total$ set. Also, both LSs will be added to the set $Buffer$, because their estimated selectivity (0.94 and 0.92 resp.) is above $sel_{des} = 0.57$. Finally, ⓪ is extracted from the $Buffer$ set. The optimization time required for this iteration was 3 seconds.

For the second iteration of the main loop of Liger, the $getNextNode(Buffer, S, T)$ function for C-RO will set to L the ② LS since it has the lowest estimated runtime in $Buffer$. Please note that for RO-MA, both ① and ② are not refined, have the same level in the refinement tree and were produced from the same recent refined LS, ⓪. As a result, RO-MA's $getNextNode(Total, Buffer, S, T)$ will also choose the same LS to refine as C-RO. Now in the main algorithm, rt_Λ will be set to 35 and Λ to ②. Then Liger will invoke the $refine(L)$ function that will fill the set $newLSs$ with ③ and ④. Since both LSs are new, they are going to be added to the $Total$ set. Also, both LSs will be added to the set $Buffer$, because their estimated selectivity (0.85 and 0.80 resp.) is above $sel_{des} = 0.57$. Finally, ② is extracted from

Table 7.1: Estimated runtime (*rt*) and selectivity (*sel*) for each subsumed LS. Estimated runtime is in seconds.

Label	LS	*rt*	*sel*
⓪	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.7), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.3))$	50	0.95
①	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.8), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.3))$	36	0.94
②	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.7), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.4))$	35	0.92
③	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.8), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.4))$	37	0.85
④	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.7), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.5))$	40	0.80
⑤	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.9), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.3))$	10	0.71
⑥	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.9), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.4))$	8	0.73
⑦	$(\epsilon, 0.6, \sqcup(\texttt{levenSim}(:\textit{label}, :\textit{label}), 0.8), (\texttt{trigrams}(\textit{title}, \textit{name}), 0.5))$	6	0.73

the $Buffer$ set. The optimization time required for this iteration was 4 seconds.

For the third iteration of the main loop of Liger, the $getNextNode(Buffer, S, T)$ function for C-RO will set to L the ① LS since it has the lowest estimated runtime in $Buffer$. For RO-MA, the $getNextNode(Total, Buffer, S, T)$ function will choose the next LS to refine based on the last refined LS, ②. As a result, LS ③ will be chosen since it has the lowest estimated runtime compared to ④. Please note that ① has the same level as ② and it will not be considered as a candidate, following the *monotonicity of run times* assumption. Now in the main algorithm, rt_Λ for C-RO will be set to 36 and Λ to ①, and for RO-MA rt_Λ will be set to 37 and Λ to ③.

Then Liger will invoke the $refine(L)$ function that will fill the set *newLSs* with ⑤ and ③ for C-RO, and with ⑥ and ⑦ for RO-MA. For C-RO, only ⑤ will be added to the $Total$ and $Buffer$ sets since ③ has been introduced in the previous main loop. For RO-MA both ⑥ and ⑦ will be added to the $Buffer$ and $Total$ sets. Finally, ① and ③ will be extracted from $Buffer$ for C-RO and RO-MA. The optimization time required for this iteration will be 4 seconds and the main loop of Liger algorithm is going to be terminated. As a result, C-RO will return ① as the best subsumed LS whereas, RO-MA will choose ③.

7.3 Evaluation

7.3.1 Evaluation Questions

The aim of our evaluation was to address the following questions:

- Q_1: Is the combined run time of the search for the best subsumed LS Λ and the execution of said LS more time-efficient than the execution of the LS L_0?
- Q_2: To what extent do the different values of *maxOpt* influence the overall run time of partial-recall Link Discovery?
- Q_3: How do the strategies C-RO and RO-MA compare to each other?
- Q_4: How much does the sampling of links influence supervised machine learning for Link Discovery?

7.3.2 Evaluation Datasets

We evaluated our approach on seven datasets. The first four were the benchmark datasets for Link Discovery dubbed Abt-Buy, Amazon-GP, DBLP-ACM and DBLP-Scholar described in [103]. These are manually curated benchmark datasets collected from real data sources such as the publication sites DBLP and ACM. We used three additional datasets (MOVIES, TOWNS and VILLAGES) from the real datasets DBpedia[3], LinkedGeodata[4] and LinkedMDB[5] to explore the scalability of the approach presented herein.[6] Information about the characteristics of the datasets and the LSs used during our experiments can be found in Table 7.2.

Table 7.2: Entity matching characteristics of datasets

Dataset	**Source (S)**	**Target (T)**	$\lvert S\rvert \times \lvert T\rvert$	**Source Property**	**Target Property**
Abt-Buy	Abt	Buy	1.20×10^6	product name description manufacturer price	product name description manufacturer price
Amazon-GP	Amazon	Google Products	4.40×10^6	product name description manufacturer price	product name description manufacturer price
DBLP-ACM	ACM	DBLP	6.00×10^6	title, authors venue, year	title authors venue, year
DBLP-Scholar	DBLP	Google Scholar	0.17×10^9	title, authors venue, year	title, authors venue, year
MOVIES	DBpedia	LinkedMDB	0.17×10^9	dbp:name dbo:director/dbp:name dbo:producer/dbp:name dbp:writer/dbp:name rdfs:label	dc2:title movie:director/movie:director_name movie:producer/movie:producer_name movie:writer/movie:writer_name rdfs:label
VILLAGES	DBpedia	LGD	6.88×10^9	rdfs:label dbo:populationTotal geo:geometry	rdfs:label lgdo:population geom:geometry/agc:asWKT

7.3.3 Experimental Setup

All LSs used during our experiments were generated automatically by the unsupervised version of the genetic-programming-based ML approach Eagle [141], as implemented in LIMES [137]. The number of generations and population size was set to 20, mutation and crossover rates were set to 0.6. Unsupervised Eagle constructed 100 independent LSs for the datasets Abt-Buy, Amazon-GP, DBLP-ACM and MOVIES. For the datasets DBLP-Scholar, TOWNS and VILLAGES, Eagle was executed 5 times, and for each run we selected the output of all 20 generations. All experiments were carried out on a 40-core Linux server running *OpenJDK* 64-Bit server 1.8.0.66 on Ubuntu 14.04.3 LTS with Intel(R) Xeon(R) CPU E5-2650 v3 processors clocked at 2.3GHz. Each experiment was repeated three times. We report the minimum run times of each of the algorithms.

We conducted partial-recall experiments with all seven datasets described previously to answer $Q_1 - Q_4$. We set the values of k to 0.1, 0.2 and 0.5 while the maximum times (*maxOpt*) for finding a partial-recall LS were set to 100, 200, 400, 800 and 1600 ms.

Finally, in all of our experiments, we used the open-source LD framework Limes as reference framework. Limes was used to execute both the input LS L_0 and the partial-recall LS Λ. The results achieved with L_0 were our *Baseline*. We used Limes as our baseline since it has

[3]`http://wiki.dbpedia.org/`

[4]`http://linkedgeodata.org/`

[5]`http://www.linkedmdb.org/`

[6]The new datasets, as well as a description of how they were constructed and the full set of results are available at `https://hobbitdata.informatik.uni-leipzig.de/LIGER/`.

outperformed state-of-the-art approaches in previous publications [137]. The main goal of running *Baseline* was to compare Liger's performance with a LD strategy that is not influenced by time and selectivity constraints.

7.3.4 Experiments Results

Scalability Improvement through Partial Recall

To answer the questions mentioned in Section 7.3.1, we evaluated the performance of Liger (i.e., C-RO and RO-MA) in terms of execution time (see Table 7.3). For the sake of comparison, we present results of the *Baseline* for all seven datasets alongside Liger. As expected, all variations of Liger require less execution time than *Baseline*. This clearly answers our first question Q_1: Liger produces more time-efficient LSs, even when *maxOpt* is set to a high value. Table 7.3 also shows clearly that Liger performs best on VILLAGES for $k = 0.1$ and $maxOpt = 0.4\,s$, where it can reduce the average run time of the 100 LSs we considered by 88%. On the smaller BDLP-ACM dataset, RO-MA performs best and achieves a time reduction of the run time by 77.5%.

Figures 7.2, 7.3 and 7.4 illustrate an initial LS and the subsumed LSs obtained by executing C-RO and RO-MA resp. for the Abt-Buy dataset, for $k = 0.1$ and $maxOpt = 100$. The resulting subsumed LSs for both Liger alternative methods require 402 ms and 130 ms to be executed, compared to the *Baseline* method that requires 753 ms. This observation supports our aforementioned conclusion that Liger is able to achieve a better performance than the *Baseline* method.

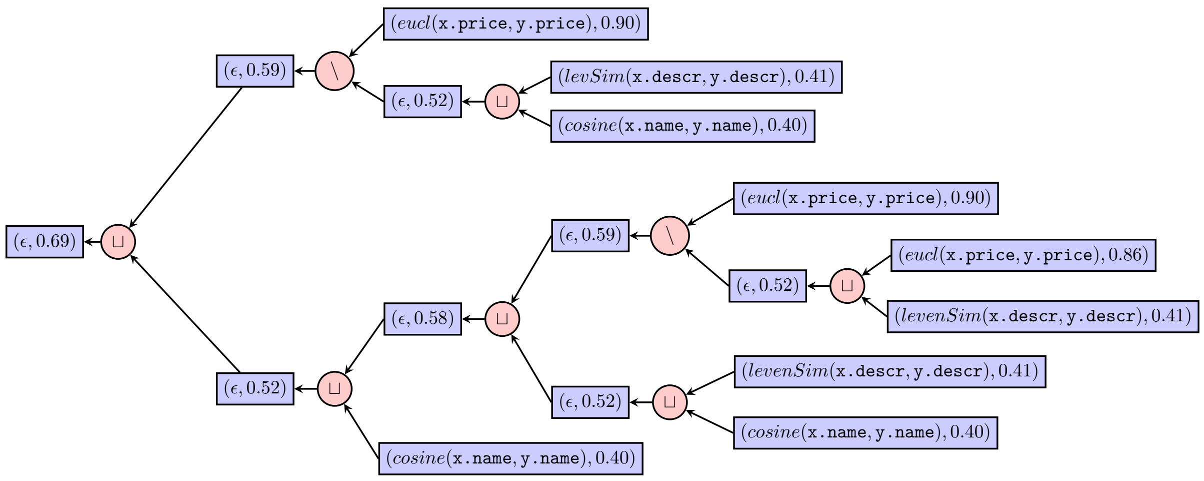

(eucl(x.price, y.price), 0.90)
(ϵ, 0.59)
\
(levSim(x.descr, y.descr), 0.41)
(ϵ, 0.52)
⊔
(cosine(x.name, y.name), 0.40)
(eucl(x.price, y.price), 0.90)
(ϵ, 0.69)
⊔
(ϵ, 0.59)
\
(eucl(x.price, y.price), 0.86)
(ϵ, 0.52)
⊔
(levenSim(x.descr, y.descr), 0.41)
(ϵ, 0.58)
⊔
(levenSim(x.descr, y.descr), 0.41)
(ϵ, 0.52)
⊔
(ϵ, 0.52)
⊔
(cosine(x.name, y.name), 0.40)
(cosine(x.name, y.name), 0.40)

Figure 7.2: Tree of a LS produced by Eagle for the Abt-Buy dataset. Note that we use `desc` for *description*, `eucl` for *euclidean* and `levenSim` for the Levenshtein similarity.

Table 7.3 also allows us to study the influence of *maxOpt* and k on total runtimes. The behavior of our approaches on the datasets varies and depends on a combination of the size of LSs (smaller LSs are easier to optimize and execute), and the difference between the execution times of the optimized specifications and the original specification. For the Amazon-GP, DBLP-Scholar, MOVIES and VILLAGES datasets, we notice that for the same value of *maxOpt*, the runtime of both RO-MA and C-RO increases as k receives larger values. This is due to the LS set for these datasets consisting mostly of large complex LSs. For the DBLP-ACM and TOWNS datasets, we noticed that the behavior of Liger is not highly influenced by the different combinations of our parameters. The observation is based on the fact that the set of LSs for both datasets consists of a more balanced proportion of atomic and complex LSs, where

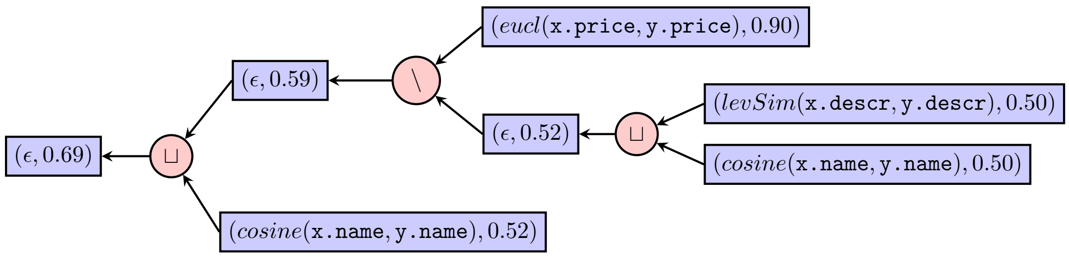

$(eucl(\texttt{x.price}, \texttt{y.price}), 0.90)$
$(\epsilon, 0.59)$
$\setminus$
$(levSim(\texttt{x.descr}, \texttt{y.descr}), 0.50)$
$(\epsilon, 0.52)$
$\sqcup$
$(cosine(\texttt{x.name}, \texttt{y.name}), 0.50)$
$(\epsilon, 0.69)$
$\sqcup$
$(cosine(\texttt{x.name}, \texttt{y.name}), 0.52)$

Figure 7.3: Tree of a LS produced by C-RO for the Abt-Buy dataset. Note that we use `desc` for *description*, `eucl` for *euclidean* and `levenSim` for the Levenshtein similarity.

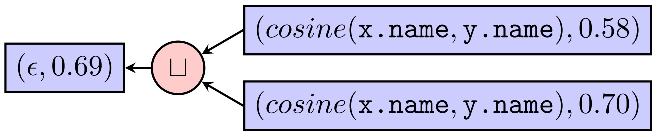

$(cosine(\texttt{x.name}, \texttt{y.name}), 0.58)$
$(\epsilon, 0.69)$
$\sqcup$
$(cosine(\texttt{x.name}, \texttt{y.name}), 0.70)$

Figure 7.4: Tree of a Link Specification produced by RO-MA for the Abt-Buy dataset

the complex LSs are not large in size compared to the previous 4 datasets. And finally, for the Abt-Buy dataset, both RO-MA and C-RO have a less uniform performance for the different values of k and *maxOpt*. The Abt-Buy's LS set consists mostly of atomic LSs. Consequently, the runtimes of the specifications are lowest for $k = 0.2$. Overall, $maxOpt = 200\,ms$ produces the best execution times on average for both C-RO and RO-MA.

To address Q_3, we studied the overall run times for all experimental configurations (see Table 7.3). Our average results suggest that RO-MA outperforms C-RO. The statistical significance of these results is confirmed by a paired t-test on the average run time distributions (significance level = 0.95). Our intuition that the *monotonicity of run times* can potentially improve the run time of our approach is supported by the results on three out of the seven datasets (Abt-Buy, DBLP-ACM and Amazon-GP). On the remaining four datasets, RO-MA outperforms C-RO on average. Still, when C-RO outperforms RO-MA, the absolute differences are minute. Figures 7.3 and 7.4 show that the subsumed LSs obtained by RO-MA outperform the refined LSs obtained by C-RO, since RO-MA creates 3 times less execution overhead for $maxOpt = 100$. Additionally, both subsumed LSs received the same selectivity. Hence, when the available refinement time is limited, RO-MA should be preferred when aiming to carry out partial-recall LD.

The highest absolute difference between C-RO and RO-MA is achieved on the DBLP-Scholar dataset, where RO-MA is 1179.59 s faster than C-RO, while the highest relative gain of 776.28% by C-RO against *Baseline* is achieved on VILLAGES (k=10%, $maxOpt = 400$), which is the largest dataset of our experiments. A particularity of DBLP-Scholar is that the LSs generated by Eagle are large, which leads to small optimization times being sufficient to find good specifications. The different strategies followed by C-RO and RO-MA lead to different atomic measures being modified during the refinement process. Especially for DBLP-Scholar, RO-MA achieves this highest absolute difference because RO-MA is able to find subsumed LSs for complex LSs more efficiently. Overall, we can answer Q_3 by stating that RO-MA is to be preferred over C-RO.

We also studied how well our expected recall approximates real recall (see Figures 7.5- 7.11). In most cases, our approximation correlates well with the real recall achieved by the specifications (see, e.g., Figures 7.5, 7.6, 7.7, 7.9). Table 7.4 shows the root mean square error (RMSE) for *Baseline*, C-RO and RO-MA between the estimated selectivity and real selectivity obtained by executing our 100 LSs. It is obvious that the RMSE between the real and estimated values is

Table 7.3: Average execution times of *Baseline*, C-RO and RO-MA for the different combinations of k and *maxOpt* over 100 LSs per dataset. All times are in seconds.

		Abt-Buy			Amazon-GP			DBLP-ACM			DBLP-Scholar		
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**
$k = 0.1$	0.1	0.66	0.52	0.52	5.71	3.91	3.82	1.08	0.25	0.25	792.81	596.53	598.44
	0.2	0.66	0.55	0.54	5.71	3.81	2.89	1.08	0.26	0.26	792.81	545.22	546.01
	0.4	0.66	0.45	0.44	5.71	3.04	2.91	1.08	0.26	0.25	792.81	589.72	587.87
	0.8	0.66	0.55	0.53	5.71	3.27	3.15	1.08	0.28	0.26	792.81	598.54	599.03
	1.6	0.66	0.54	0.51	5.71	3.47	3.18	1.08	0.33	0.28	792.81	554.82	557.06
		MOVIES			TOWNS			VILLAGES					
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**			
	0.1	4.05	1.89	1.89	44.52	31.15	31.20	123.58	15.32	15.26			
	0.2	4.05	1.75	1.76	44.52	32.23	32.19	123.58	15.65	15.71			
	0.4	4.05	1.91	1.90	44.52	34.21	34.09	123.58	14.10	14.12			
	0.8	4.05	1.77	1.76	44.52	34.08	34.10	123.58	14.69	14.52			
	1.6	4.05	1.93	1.89	44.52	34.38	34.00	123.58	15.65	15.17			
		Abt-Buy			Amazon-GP			DBLP-ACM			DBLP-Scholar		
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**
$k = 0.2$	0.1	0.66	0.35	0.35	5.71	4.07	3.92	1.08	0.26	0.26	792.81	593.66	581.86
	0.2	0.66	0.34	0.34	5.71	3.38	3.23	1.08	0.26	0.26	792.81	590.91	587.89
	0.4	0.66	0.29	0.28	5.71	3.38	3.36	1.08	0.29	0.28	792.81	561.93	566.74
	0.8	0.66	0.33	0.32	5.71	3.17	3.15	1.08	0.29	0.26	792.81	579.14	580.28
	1.6	0.66	0.40	0.37	5.71	3.57	3.37	1.08	0.33	0.29	792.81	551.14	549.08
		MOVIES			TOWNS			VILLAGES					
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**			
	0.1	4.05	1.98	1.99	44.52	34.49	34.42	123.58	20.60	20.64			
	0.2	4.05	1.99	1.99	44.52	32.28	32.36	123.58	20.01	19.98			
	0.4	4.05	2.01	1.98	44.52	32.56	32.46	123.58	19.40	19.42			
	0.8	4.05	1.97	1.97	44.52	33.65	33.79	123.58	21.23	20.97			
	1.6	4.05	1.99	1.99	44.52	33.98	34.04	123.58	21.15	20.80			
		Abt-Buy			Amazon-GP			DBLP-ACM			DBLP-Scholar		
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**
$k = 0.5$	0.1	0.66	0.42	0.42	5.71	4.55	4.23	1.08	0.28	0.28	792.81	602.49	603.92
	0.2	0.66	0.44	0.43	5.71	4.02	3.64	1.08	0.28	0.28	792.81	554.38	555.24
	0.4	0.66	0.42	0.41	5.71	3.95	3.65	1.08	0.27	0.27	792.81	637.66	629.34
	0.8	0.66	0.42	0.41	5.71	3.66	3.63	1.08	0.29	0.27	792.81	590.48	581.50
	1.6	0.66	0.48	0.45	5.71	3.79	3.67	1.08	0.34	0.29	792.81	595.17	591.91
		MOVIES			TOWNS			VILLAGES					
	maxOpt	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**	**Baseline**	**C-RO**	**RO-MA**			
	0.1	4.05	2.83	2.81	44.52	32.05	31.96	123.58	25.50	25.43			
	0.2	4.05	2.81	2.78	44.52	34.79	34.59	123.58	29.231	29.23			
	0.4	4.05	2.81	2.84	44.52	34.57	34.08	123.58	25.60	25.58			
	0.8	4.05	2.92	2.89	44.52	31.84	31.87	123.58	29.21	29.17			
	1.6	4.05	2.91	2.91	44.52	34.69	34.57	123.58	28.49	28.52			

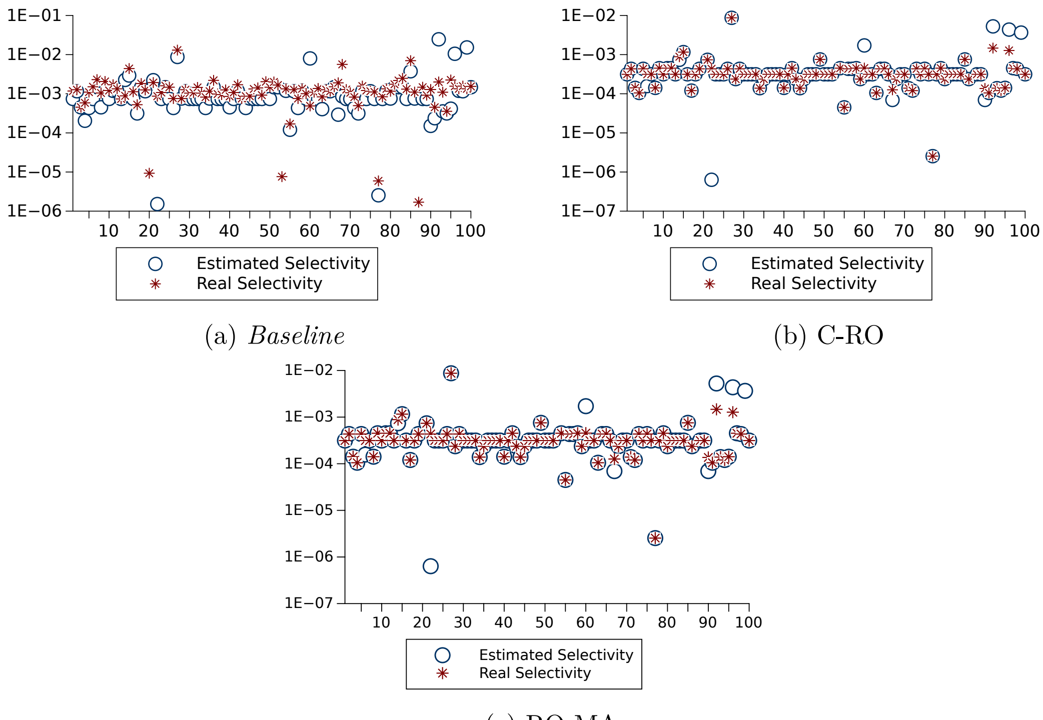
rather small. Consequently, our results suggest that our recall estimation function is reliable. The dataset on which our expected recall is the furthest away from the real recall is the DBLP-Scholar dataset. We assume that this is due to the heterogeneous distribution of characters in the datasets, which leads to a poor approximation of intermediate results. We will investigate this behavior in future works.

Applications to Supervised Learning

As pointed out in our introduction, one of the main reasons for using partial-recall Link Discovery is its application in areas of machine learning where approaches must be executed on samples due to time constraints, or where seeding is regarded as viable during the initialization phase of the algorithms. One of these areas is link discovery itself, as predictive maintenance approaches learn link specifications to integrate data across a whole industrial plant. In this extrinsic evaluation, we aim to measure the loss of F-measure of a machine-learning approach when presented with

Table 7.4: Root mean square error (RMSE) for *Baseline*, C-RO and RO-MA between the estimated selectivity and real selectivity for $k = 0.2$ and $maxOpt = 1600$.

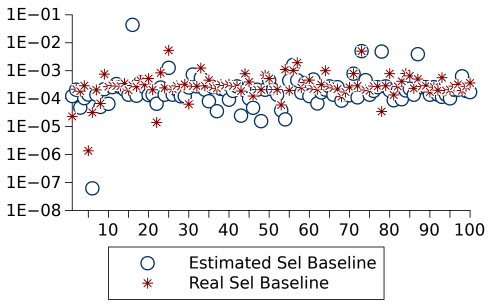
Dataset	**RMSE for *Baseline***	**RMSE for C-RO**	**RMSE for RO-MA**
Abt-Buy	3.04E-04	6.26E-05	6.26E-05
Amazon-Google	4.41E-04	4.24E-04	4.23E-04
DBLP-ACM	7.40E-04	7.47E-05	1.25E-04
DBLP-Scholar	6.29E-04	1.59E-04	1.59E-04
MOVIES	1.22E-04	4.05E-05	4.05E-05
TOWNS	1.18E-05	4.03E-06	4.03E-06
VILLAGES	1.44E-06	2.16E-07	2.16E-07



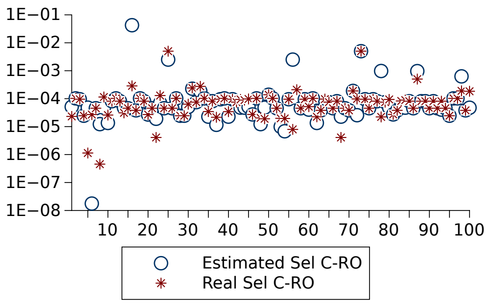
(a) *Baseline* (b) C-RO (c) RO-MA

Figure 7.5: Real selectivity and estimated selectivity results for *Baseline*, C-RO and RO-MA on Abt-Buy data. The x-axis represents the number of specifications, the y-axis represents the selectivity in logarithmic scale for $k = 0.2$ and $maxOpt = 1600$.

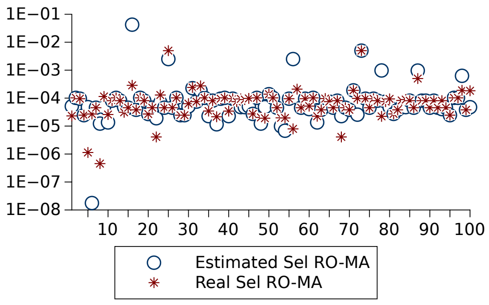
the results of partial-recall Link Discovery in comparison with the F-measure it would achieve using the full results. To this end we use Wombat, which is currently the only approach for learning LSs from positive examples. Figure 7.12 shows the results achieved by Wombat when presented with the complete set of links generated by Eagle, as well as the results generated by partial linking. Our results clearly show that with an expected partial recall of 50%, Wombat achieves at least 76.6% of the F-measure that it achieves when presented with all the data generated by Eagle (recall = 100%). As expected, the overall loss in F-measure is on datasets where the achievable F-measure is smaller, (e.g., on Amazon-GP). Still, in the best case, we achieve more than 95% of the maximal F-measure with $k = 0.1$. This gives us a clear answer to Q_4, namely that while the sampling of links leads to smaller F-measures, the ratio between expected recall and portion of F-measure achieved speaks in favor of using partial-recall Link Discovery in machine-learning applications with time constraints.



(a) *Baseline*

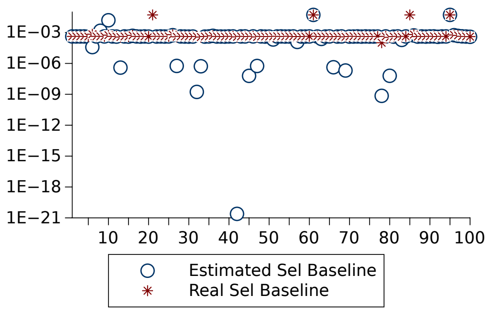


(b) C-RO

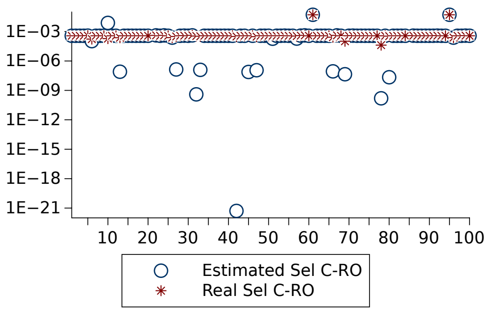


(c) RO-MA

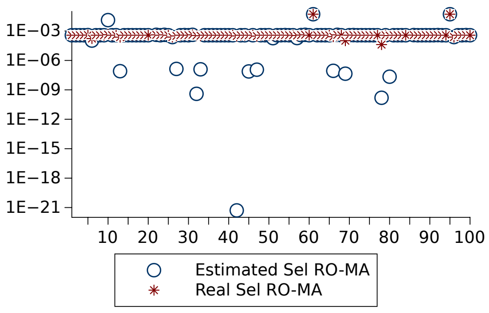
Figure 7.6: Real selectivity and estimated selectivity results for *Baseline*, C-RO and RO-MA on Amazon-GP data. The x-axis represents the number of specifications, the y-axis represents the selectivity in logarithmic scale for $k = 0.2$ and $maxOpt = 1600$.



(a) *Baseline*

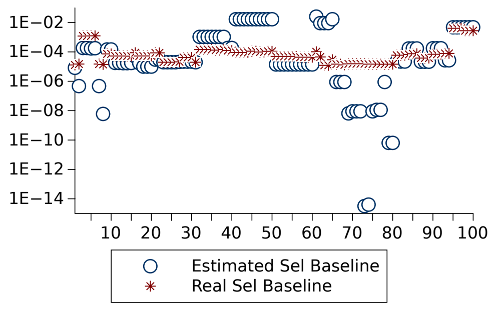


(b) C-RO

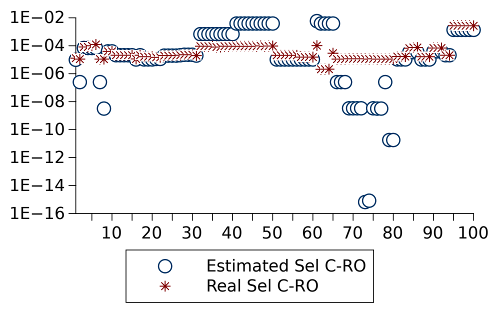


(c) RO-MA

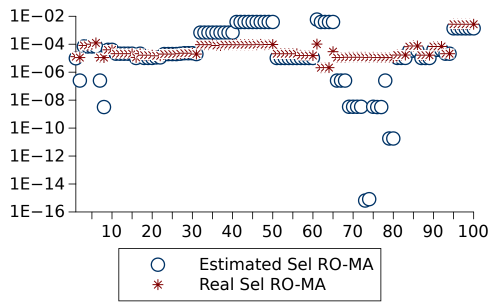
Figure 7.7: Real selectivity and estimated selectivity results for *Baseline*, C-RO and RO-MA on DBLP-ACM data. The x-axis represents the number of specifications, the y-axis represents the selectivity in logarithmic scale for $k = 0.2$ and $maxOpt = 1600$.



(a) *Baseline*

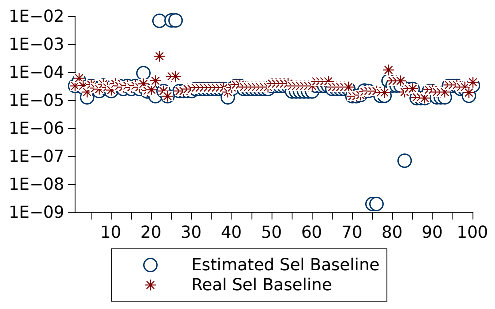


(b) C-RO

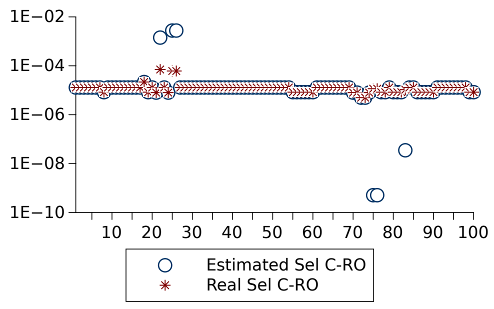


(c) RO-MA

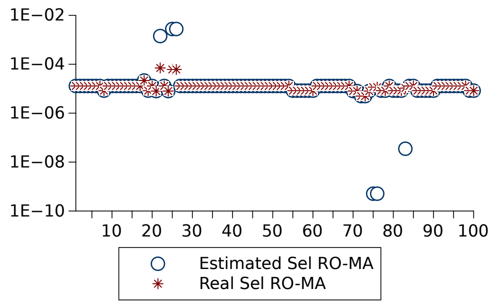
Figure 7.8: Real selectivity and estimated selectivity results for *Baseline*, C-RO and RO-MA on DBLP-Scholar data. The x-axis represents the number of specifications, the y-axis represents the selectivity in logarithmic scale for $k = 0.2$ and $maxOpt = 1600$.



(a) *Baseline*

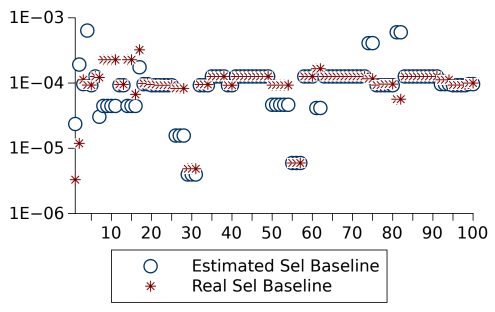


(b) C-RO

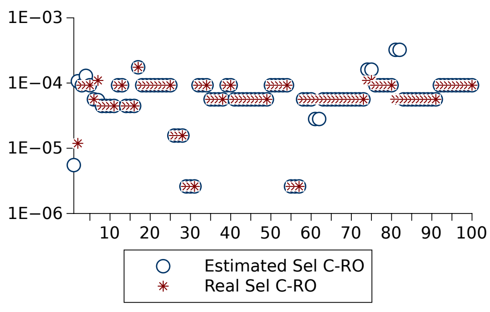


(c) RO-MA

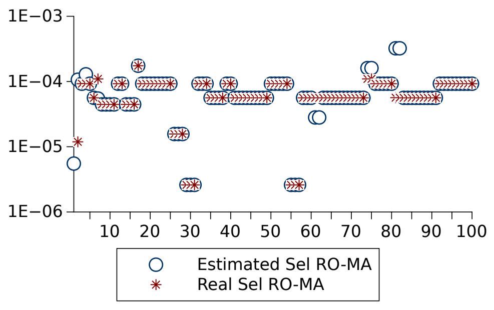
Figure 7.9: Real selectivity and estimated selectivity results for *Baseline*, C-RO and RO-MA on MOVIES data. The x-axis represents the number of specifications, the y-axis represents the selectivity in logarithmic scale for $k = 0.2$ and $maxOpt = 1600$.



(a) *Baseline*

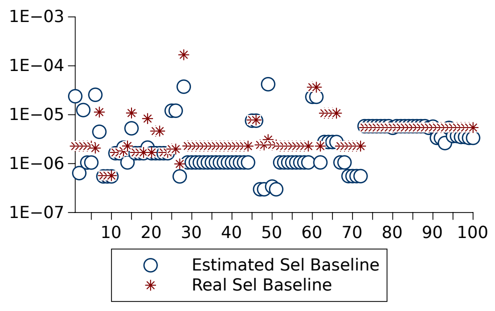


(b) C-RO

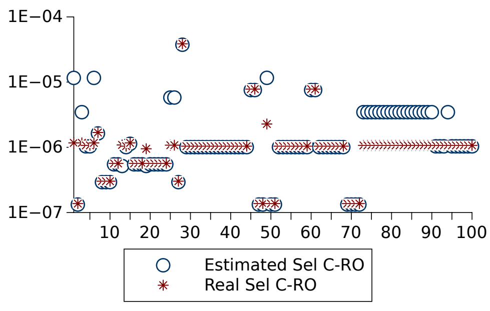


(c) RO-MA

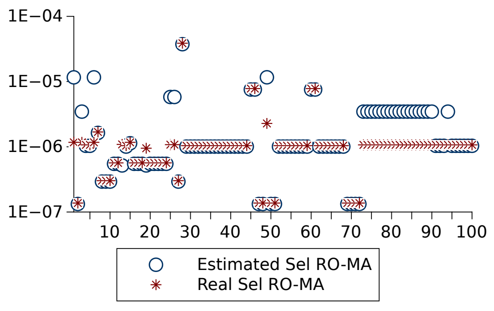
Figure 7.10: Real selectivity and estimated selectivity results for *Baseline*, C-RO and RO-MA on TOWNS data. The x-axis represents the number of specifications, the y-axis represents the selectivity in logarithmic scale for $k = 0.2$ and $maxOpt = 1600$.



(a) *Baseline*



(b) C-RO



(c) RO-MA

Figure 7.11: Real selectivity and estimated selectivity results for *Baseline*, C-RO and RO-MA on VILLAGES data. The x-axis represents the number of specifications, the y-axis represents the selectivity in logarithmic scale for $k = 0.2$ and $maxOpt = 1600$.

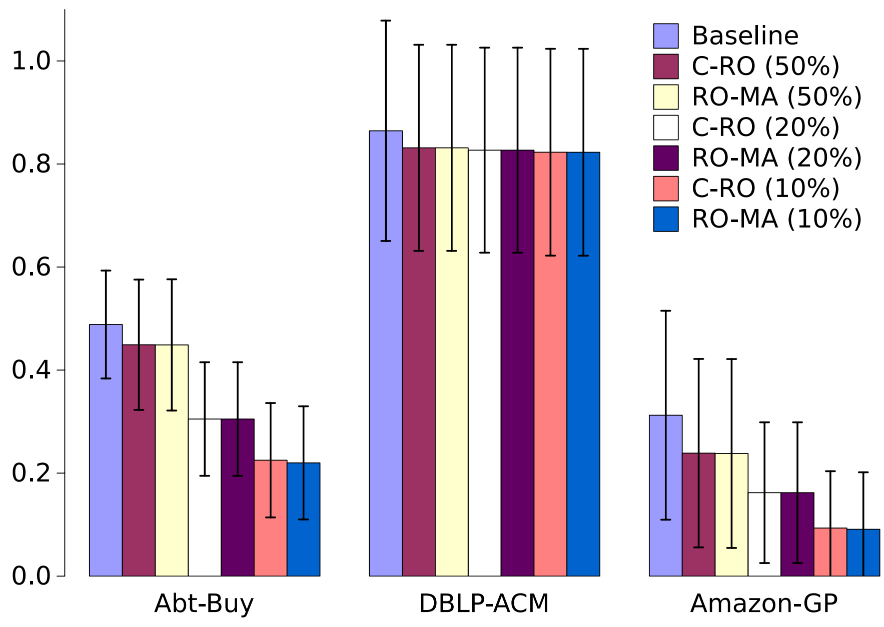


Figure 7.12: F-measures achieved by Wombat, when provided with the results of Liger as training data. The expected recall values set for C-RO and RO-MA are in brackets.

8 Dynamic Planning for Link Discovery

Preamble This chapter is based on [65] which is the first work on dynamic planning for LD. The author co-designed, implemented and evaluated the algorithm presented herein, and co-wrote the said paper.

8.1 The CONDOR Approach

The goal of CONDOR (DynamiC Planning fOr LiNk DiscOveRy) is to improve the overall execution time of LSs. To this end, CONDOR aims to derive a time-efficient execution plan for a given input LS L. The basic idea behind state-of-the-art planners for LD (see [138]) is to approximate the costs of possible plans for L, and to simply select the least costly (i.e., the presumable fastest) plan to improve the execution costs. The selected plan is then forwarded to the execution engine and executed. We call this type of planning *static planning* because the plan selected is never changed. CONDOR addresses the planning and execution of LSs differently: given an input LS L, CONDOR's planner uses an initial cost function to generate initial plans, of which each consists of a sequence of steps that are to be executed by CONDOR's execution engine to compute L. The planner chooses the least costly plan and forwards it to the engine. After the execution of each step, the execution engine overwrites the planner's cost function by replacing the estimated costs of the executed step with its real costs. The planner then re-evaluates the alternative plans previously generated, and alters the remaining steps to be executed if the updated cost function suggests better expected runtimes for this alteration of the remaining steps. We call this novel paradigm for planning the execution of LSs *dynamic planning.*

8.1.1 Planning

Algorithm 17 summarizes the dynamic planning approach implemented by CONDOR. The algorithm (dubbed *plan*) takes a LS L as input and returns the plan P with the smallest expected runtime. The core of the approach consists of (1) a cost function re, which computes expected runtimes as described in Chapter 6, and (2) a recursive cost evaluation scheme. CONDOR's planner begins by retrieving a map with LSs as keys and their corresponding plans as values (line 2).[1] Then it checks whether the input L has already been executed within the current run (line 3). If L has already been executed, there is no need to re-plan the LS. Instead, *plan* returns

[1]The map returned from function $getExecutedPlans()$ has global scope and is updated in the $execute(P, S, T)$ function

the known plan P. If L has not yet been executed, we proceed by first checking whether L is atomic. If L is atomic, we return $P = run(m(p_s, p_t), \theta)$ (line 7), which simply computes $[[L]]$ on $S \times T$. Here, we make use of existing scalable solutions for computing such mapping [134].

If $L = (f, \zeta, \omega(L_1, L_2))$, *plan* derives a plan for L_1 and L_2 (lines 11 and 12), then computes the possible plans given *op*(L). It then decides for the least costly plan based on the cost function. The possible plans generated by Condor depend on the operator of L. For example, if *op*$(L) = \sqcap$, then *plan* algorithm evaluates three alternative plans:

1. The *canonical* plan (lines 22, 24, 28, 32), which consists of executing the plans of L_1 and L_2 (P_1 and P_2 resp.), performing an intersection between the resulting mappings and then filtering the final mapping using (f, ζ);
2. The *filter-right* plan (lines 25, 33), where the best plan P_1 for L_1 is executed, followed by a filtering operation run on the results of P_1 using $(f_2, \zeta_2) = \varphi(L_2)$. Then the final mapping is filtered using (f, ζ);
3. The *filter-left* plan (lines 29, 33), which is a *filter-right* plan with the roles of L_1 and L_2 reversed.

Condor's planning function re-uses results of previously executed LSs and plans. Hence, if both P_1 and P_2 have already been executed ($re(P_1) = re(P_2) = 0$), then the best plan is the *canonical* plan, where Condor will only need to retrieve the mappings of the two plans and then perform the intersection and the filtering operation (line 21). If P_1 resp. P_2 have already been executed (see line 23 resp. 27), then the algorithm decides between the *canonical* and the *filter-right* resp. *filter-left* plan. If no information is available, then the costs of the different alternatives are calculated based on our cost function described in Section 8.1.2 and the least costly plan is chosen. Similar approaches are implemented for *op*$(L) = \setminus$ (lines 13- 19). In particular, in line 18, the *plan* algorithm implements the *filter-right* plan by first executing the plan P_1 for the left child and then constructing a "reverse filter" from $(f_2, \tau_2) = \varphi(L_2)$ by calling the *getReverseFilter* function. The resulting filter is responsible for only allowing links of the retrieved mapping of L_1 not returned by L_2. For *op*$(L) = \sqcup$ (line 37) the plan always consists of merging the results of P_1 and P_2.

8.1.2 Plan Evaluation

One important component of Condor is the cost function required to estimate the costs of executing the corresponding plan. Based on [64] and Section 6, we used a linear plan evaluation schema as introduced in [138]. A plan P is characterized by one basic component, $re(P)$, the approximated runtime of executing P.

Approximation of $re(P)$ for an atomic LS

We compute $re(P)$ by assuming that the runtime of $L = (m(p_s, p_t), \theta)$ can be approximated in linear time for each metric m using the following equation:

$$re(P) = \kappa_0 + \kappa_1|S| + \kappa_2|T| + \kappa_3\theta \quad , \tag{8.1}$$

where $|S|$ is the size of the source KB, $|T|$ is the size of the target KB and θ is the threshold of the specification. The next step of our plan evaluation approach was to estimate the parameters $\kappa_0, \kappa_1, \kappa_2and\kappa_3$. However, the size of the source and target KBs was unknown prior to the linking task. Therefore, we used a sampling method, where we generated source and target datasets of sizes $1,000, 2,000, \ldots, 10,000$ by sampling data from the English labels of DBpedia 3.8. and

Algorithm 17: *plan* Algorithm for Condor

```
Input: a link specification $L$
Output: Least costly plan $P$ of $L$
1  $P \leftarrow \emptyset$
2  $executedPlans \leftarrow getExecutedPlans()$
3  if $executedPlans.contains(L)$ then
4      $P \leftarrow executedPlans.get(L)$
5  else
6      if $(L == (m(p_s, p_t), \theta))$ then
7          $P \leftarrow run(m(p_s, p_t), \theta)$
8      else
9          $L_1 \leftarrow L.leftChild$
10         $L_2 \leftarrow L.rightChild$
11         $P_1 \leftarrow plan(L_1)$
12         $P_2 \leftarrow plan(L_2)$
13         if $(op(L) == \backslash)$ then
14             if $executedPlans.contains(L_2)$ then
15                 $P \leftarrow merge(minus, P_1, P_2)$
16             else
17                 $Q_0 \leftarrow merge(minus, P_1, P_2)$
18                 $Q_1 \leftarrow merge(getReverseFilter(\varphi(L_2)), P_1)$
19                 $P \leftarrow getLeastCostly(Q_0, Q_1)$
20         else if $(op(L) == \sqcap)$ then
21             if $(executedPlans.contains(L_1) \wedge executedPlans.contains(L_2))$ then
22                 $P \leftarrow merge(intersection, P_1, P_2)$
23             else if $(executedPlans.contains(L_1) \wedge \neg executedPlans.contains(L_2))$ then
24                 $Q_0 \leftarrow merge(intersection, P_1, P_2)$
25                 $Q_1 \leftarrow merge(\varphi(L_2), P_1)$
26                 $P \leftarrow getLeastCostly(Q_0, Q_1)$
27             else if $(\neg executedPlans.contains(L_1) \wedge executedPlans.contains(L_2))$ then
28                 $Q_0 \leftarrow merge(intersection, P_1, P_2)$
29                 $Q_1 \leftarrow merge(\varphi(L_1), P_2)$
30                 $P \leftarrow getLeastCostly(Q_0, Q_1)$
31             else
32                 $Q_0 \leftarrow merge(intersection, P_1, P_2)$
33                 $Q_1 \leftarrow merge(\varphi(L_2), P_1)$
34                 $Q_2 \leftarrow merge(\varphi(L_1), P_2)$
35                 $P \leftarrow getLeastCostly(Q_0, Q_1, Q_2)$
36         else
37             $P \leftarrow merge(union, P_1, P_2)$
38 Return $P$
```

stored the runtime of the measures implemented by our framework for different thresholds θ between 0.5 and 1. Then, we computed the κ_i parameters by deriving the solution of the problem to the linear regression solution of $K = (\Pi^T\Pi)^{-1}\Pi^T\Xi$, where $K = (\kappa_0, \kappa_1, \kappa_2, \kappa_3)^T$, Ξ is a vector in which the ξ_i-th row corresponds to the runtime retrieved by running ith experiment and Π is a four-column matrix in which the corresponding experimental parameters $(1, |S|, |T|, \theta)$ are stored in the π_i-th row.

Approximation of $re(P)$ for a complex LS

For the *canonical* plan, $re(P)$ is estimated by summing up the runtime estimations of all plans that correspond to children specifications of the complex LS. For the *filter-right* and *filter-left* plans, $re(P)$ is estimated by summing up the $re(P_i), i = 1, 2$ of the children LS whose plan is to be executed along with the filtering function runtime approximation performed by the other child LS. To estimate the runtime of a filtering function, we compute the approximation analogously to the computation of the runtime of an atomic LS.

Additionally, we define a set of rules if $\omega = \sqcap$ or $\omega = \backslash$:

1. $re(P)$ includes only the sums of the children LSs that have not yet been executed.
2. If both children of the LS are executed then $re(P)$ is set to 0. Therefore, we force the algorithm to choose *canonical* over the other two options, since it will create a smaller overhead in total runtime of Condor.

8.1.3 Execution

Algorithm 18 describes the execution of the plan that Algorithm 17 returned. The *execute* algorithm takes as input a LS L, a source KB S and a target KB T, and returns the corresponding mapping M once all steps of P have been executed. The algorithm begins in line 4, where *execute* returns the mapping M of L from the map *results*, if L has already been executed and its result cached.[2] If L has not been executed before, we proceed by checking whether a LS L' with $[[L]] \subseteq [[L']]$ has already been executed (line 8). If such a L' exists, then *execute* retrieves $M' = [[L']]$ and runs $(f, \zeta, [[L']])$ where $(f, \zeta) = \varphi(L)$ (line 10). If such a L' does not exist, the algorithm checks whether L is atomic. If this is the case, it calls the function $plan(L)$ (line 13) that returns the plan $P = run(m(p_s, p_t), \theta)$ and then it computes $[[L]]$. If $L = (f, \zeta, \omega(L_1, L_2))$, *execute* calls the *plan* function described previously.

8.1.4 Example Run

To elucidate the workings of Condor further, we use the LS described in Figure 8.1 as a running example. Table 8.1 shows the runtime cost returned by the function *re* for each possible plan that can be produced for the specifications included in L, for the different calls of the *plan* function for L. The runtime value of a plan for a complex LS additionally includes a value for the filtering or set operations, wherever present. Recall that *plan* is a recursive function (lines 11, 12) and plans L in post-order (bottom-up, left-to-right). Condor produces a plan equivalent to the *canonical* plan for the left child due to the $\sqcup$ operator. Then, it proceeds in finding the least costly plan for the right child. For the right child, *plan* has to choose between the three alternatives described in Sect. 8.1.1. Table 8.1 shows the approximation runtime of each plan for $(\sqcap((\texttt{cosine}(label, label), 0.40), (\texttt{trigrams}(name, name), 0.80)), 0.50)$. The least costly plan for the right child is the *filter-left* plan, where $(\texttt{trigrams}(name, name), 0.80))$ is executed and the resulting mapping is filtered using $(\texttt{cosine}(label, label), 0.40))$, and then $(\epsilon, 0.50)$. Before proceeding to discover the best plan for L, Condor assigns an approximate runtime to each child plan of L: 3.5 s for the left child's plan P_1 and 1.5 s for the right child's plan P_2.

Once Condor has identified the best plans for both children of L, it proceeds to find the most efficient plan for L. Since both children have not been previously executed, *plan* goes to line 16. There, it has to chose between two alternative plans, i.e., the *canonical* plan with $re(P) = 6.2$ s and the *filter-right* plan with $re(P) = 5.2$ s. It is obvious that *plan* is going to

[2]The result buffer *results* has global scope.

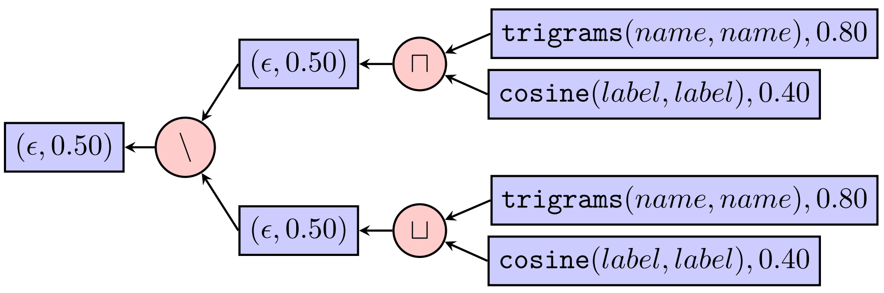
Algorithm 18: *execute* Algorithm

```
Input: a link specification $L$; a source KB $S$; a target KB $T$
Output: Mapping $M$ of $L$
1  $M \leftarrow \emptyset$
2  $executedPlans \leftarrow getExecutedPlans()$
3  $results \leftarrow getResults()$
4  if $executedPlans.contains(L)$ then
5      $M \leftarrow results.get(L)$
6  else
7      $L' = checkDependencies(L)$
8      if $L' \neq null$ then
9          $M \leftarrow results.get(L')$
10         $M \leftarrow filter(\varphi(L), M')$
11     else
12         if $L == (m(p_s, p_t), \theta)$ then
13             $P \leftarrow plan(L)$
14             $M \leftarrow run(P, S, T)$
15         else
16             $P \leftarrow plan(L)$
17             $L_1 \leftarrow P.getSubSpec(0)$
18             $M_1 \leftarrow execute(L_1, S, T)$
19             $P \leftarrow plan(L)$
20             if $P.operator \neq \emptyset$ then
21                 $L_2 \leftarrow P.getSubSpec(1)$
22                 $M_2 \leftarrow execute(L_2, S, T)$
23                 $M \leftarrow runOperator(P.operator, M_1, M_2)$
24             else
25                 if $(op(L) == \backslash)$ then
26                     $M \leftarrow filter(getReverseFilter(\varphi(L_2)), M_1)$
27                 else
28                     $M \leftarrow filter(\varphi(L_2), M_1)$
29             $M \leftarrow filter(\varphi(L), M)$
30     $update(L, P, executedPlans)$
31     $addResults(L, M)$
32 Return $M$
```

assign the *filter-right* plan as the least costly plan for L. Note that this plan overwrites the right child *filter-left* plan, and it will instead use the right child as a filter.

Once the plan is finalized, the *plan* function returns and assigns the plan shown in Figure 8.2 to $re(P)$ in line 16. For the next step, *execute* retrieves the left child $(\sqcup((\texttt{cosine}(label, label), 0.40), (\texttt{trigrams}(name, name), 0.80)), 0.50)$ and assigns it to L_1 (line 17). Then, the algorithm calls *execute* for L_1. *execute* repeats the plan procedure for L_1 recursively, and returns the plan illustrated in Fig. 8.4. The plan is executed and finally (line 18) the resulting mapping is assigned to M_1. Remember that all intermediate mappings, the final mapping, and the corresponding LSs are stored for future use (line 31). Additionally, we replace the cost value estimations of each executed plan with their real values and update the executed plans map *executedPlans* in line 30. Now, the cost value of $(\texttt{cosine}(label, label), 0.40)$ is assigned to 2.0 s, the cost value of $(\texttt{trigrams}(name, name), 0.80)$ is assigned to 1.0 s and finally, the cost value of the left child is replaced by 4.0 s.

Given the runtimes from the execution engine, the algorithm re-plans the further steps of L. Within this second call of *plan* (line 19), Condor does not re-plan the sub-specification



$(\epsilon, 0.50)$
$\setminus$
$(\epsilon, 0.50)$
$\sqcap$
trigrams(*name*, *name*), 0.80
cosine(*label*, *label*), 0.40
$(\epsilon, 0.50)$
$\sqcup$
trigrams(*name*, *name*), 0.80
cosine(*label*, *label*), 0.40

Figure 8.1: Graphical representation of an example LS

Table 8.1: Runtime costs for the plans computed for the specification in (Fig. 8.1) by the two calls of the *plan* in lines 16 and 19. All runtimes are presented in seconds. The 1^{st} column includes the initial runtime approximations of plans. The 2^{nd} column includes (1) a real runtime value of a plan, if the plan has been executed ($^\diamond$); (2) a 0.0 value if all the subsequent plans of that plan have been previously executed ($^\bullet$) or have an estimation of zero cost in the current call of *plan* (*); (3) a runtime approximation value, which includes only runtimes of subsequent plans that have not yet been executed ($^\square$).

P	$re(P)$	
	1^{st}	2^{nd}
(cosine(*label*, *label*), 0.40)	1.8	$2.0^\diamond$
(trigrams(*name*, *name*), 0.80)	0.5	$1.0^\diamond$
φ(cosine(*label*, *label*), 0.40)	0.8	$0.8^\square$
φ(trigrams(*name*, *name*), 0.80)	0.6	$0.6^\square$
canonical plan: *merge*($\sqcap$, (cosine(*label*, *label*), 0.40), (trigrams(*name*, *name*), 0.80))	3.5	$0.0^\bullet$
filter-right plan: *merge*(φ(trigrams(*name*, *name*), 0.80), (cosine(*label*, *label*), 0.40))	2.6	$0.8^\square$
filter-left plan: *merge*(φ(cosine(*label*, *label*), 0.40), (trigrams(*name*, *name*), 0.80))	1.5	$1.0^\square$
canonical plan: *merge*($\sqcup$, (cosine(*label*, *label*), 0.40), (trigrams(*name*, *name*), 0.80))	3.5	$4.0^\diamond$
canonical plan for L	6.2	0.0^*
filter-right plan for L (see Fig. 8.2)	5.2	$1.7^\square$

that corresponds to L_1, since its plan (Figure 8.4) has been executed previously. Initially, *plan* had decided to use the right child as a filter. However, both (cosine(*label*, *label*), 0.40) and (trigrams(*name*, *name*), 0.80) have already been executed. Hence, the new total cost of executing the right child is set to 0.0. Consequently, *plan* changes the remaining steps of the initial plan of L, since the cost of executing the *canonical* plan is now set to 0.0. The final plan is illustrated in Figure 8.3.

Once the new plan P is constructed, *execute* checks if P includes any operators. In our example, $op(L) = \setminus$. Thus, we execute the second direct child of L as described in P, $L_2 = (\sqcap(($cosine (*label*, *label*), 0.40), (trigrams(*name*, *name*), 0.80)), 0.50). Algorithm 18 calls the *execute* function for L_2, which calls *plan*. Condor's planning algorithm then returns a plan for L_2, which is similar to the plan for the left child illustrated in Fig. 8.4 by replacing the $\sqcup$ operator with the $\sqcap$ operator, with $re(P_2) = 0$ s.

When the algorithm proceeds to executing P_2, it discovers that the atomic LSs of L_2 have already executed. Thus, it retrieves the corresponding mappings, performs the intersection between the results of (cosine(*label*, *label*), 0.40) and (trigrams(*name*, *name*), 0.80), filters the

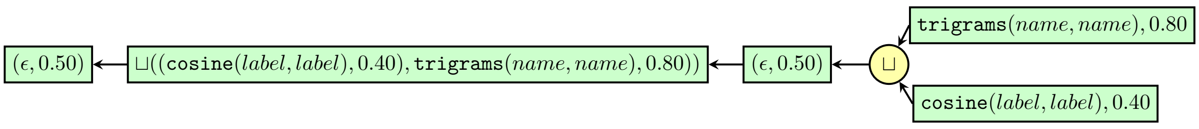

$(\epsilon, 0.50)$
$\sqcup((\texttt{cosine}(label, label), 0.40), \texttt{trigrams}(name, name), 0.80))$
$(\epsilon, 0.50)$
$\sqcup$
$\texttt{trigrams}(name, name), 0.80$
$\texttt{cosine}(label, label), 0.40$

Figure 8.2: Initial plan of LS from Figure 8.1

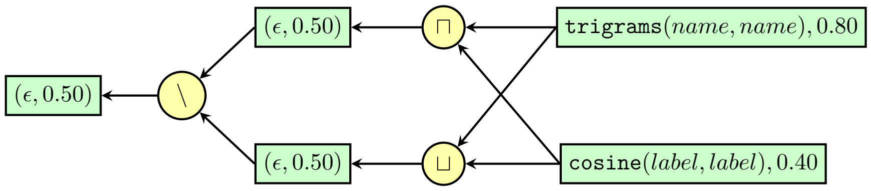


Figure 8.3: Final plan of LS from Figure 8.1

resulting mapping of the intersection with $(\epsilon, 0.50)$ and stores the resulting mapping for future use (line 31). Returning to our initial LS L, the algorithm has now retrieved results for both L_1 and L_2 and proceeds to perform the steps described in lines 23 and 29. The final plan constructed by Condor is presented in Fig. 8.3.

If the second call of the *plan* function for L in line 19 had resulted in not altering the initial plan (Figure 8.2), then *execute* would have proceeded in applying a reverse filter (i.e., the implementation of the difference of mappings) on M_1 by using $(\sqcap((\texttt{cosine}(label, label), 0.40), (\texttt{trigrams}(name, name), 0.80)), 0.50)$ (line 26). Similarly, operations would have been carried out if $op(L) = \sqcap$ in line 28.

Overall, the complexity of Condor can be derived as follows: for each node of a LS L, Condor generates a constant number of possible plans. Hence, the complexity of each iteration of Condor is $O(|L|)$. The execution engine executes at least one node in each iteration, meaning that it needs at most $O(|L|)$ iterations to execute L completely. Hence, Condor's worst-case runtime complexity is $O(|L|^2)$.

8.2 Evaluation of Condor

8.2.1 Evaluation Questions

The aim of our evaluation was to address the following questions:

- (Q_1) Does Condor achieve better runtimes for LSs?
- (Q_2) How much time does Condor spend planning?
- (Q_3) How do the different sizes of LSs affect Condor's runtime?

8.2.2 Evaluation Datasets

To evaluate the performance of Condor, we performed a set of experiments against seven data sets. The first four are benchmark data sets for LD dubbed Abt-Buy, Amazon-GP, DBLP-ACM

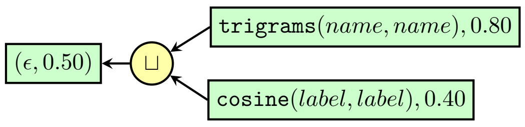

$\texttt{trigrams}(name, name), 0.80$
$(\epsilon, 0.50)$
$\sqcup$
$\texttt{cosine}(label, label), 0.40$

Figure 8.4: Plan of the left child for the LS in Fig. 8.1

and DBLP-Scholar described in [103]. These are manually curated benchmark data sets collected from real data sources, such as the publication sites DBLP and ACM, as well as the Amazon and Google product websites. To assess the scalability of Condor, we used three additional data sets (MOVIES, TOWNS and VILLAGES, see Table 8.2) from the data sets DBpedia, LGD and LinkedMDB.[3] [4] Table 8.2 describes their characteristics and presents the properties used when linking retrieved resources. The mapping properties were provided to the link discovery algorithms underlying our results.

Table 8.2: Entity matching characteristics of datasets

Dataset	**Source (S)**	**Target (T)**	$\lvert S\rvert \times \lvert T\rvert$	**Source Property**	**Target Property**
Abt-Buy	Abt	Buy	1.20×10^6	product name description manufacturer price	product name description manufacturer price
Amazon-GP	Amazon	Google Products	4.40×10^6	product name description manufacturer price	product name description manufacturer price
DBLP-ACM	ACM	DBLP	6.00×10^6	title, authors venue, year	title authors venue, year
DBLP-Scholar	DBLP	Google Scholar	0.17×10^9	title, authors venue, year	title, authors venue, year
MOVIES	DBpedia	LinkedMDB	0.17×10^9	dbp:name dbo:director/dbp:name dbo:producer/dbp:name dbp:writer/dbp:name rdfs:label	dc2:title movie:director/movie:director_name movie:producer/movie:producer_name movie:writer/movie:writer_name rdfs:label
VILLAGES	DBpedia	LGD	6.88×10^9	rdfs:label dbo:populationTotal geo:geometry	rdfs:label lgdo:population geom:geometry/agc:asWKT

8.2.3 Experimental Setup

We generated 100 LSs for each dataset by using the unsupervised version of Eagle, a genetic programming approach for learning LSs [141]. We used this algorithm because it can detect LSs of high accuracy on the datasets at hand. We configured Eagle by setting the number of generations and population size to 20; mutation and crossover rates were set to 0.6. All experiments were carried out on a 20-core Linux Server running *OpenJDK* 64-Bit Server 1.8.0.66 on Ubuntu 14.04.3 LTS on Intel Xeon CPU E5-2650 v3 processors clocked at 2.30GHz. Each experiment was repeated three times. We report the average runtimes of each of the algorithms. Note that all three planners returned the same set of links and that they hence all achieved 100% F-measure w.r.t. the LS to be executed.[5]

8.2.4 Evaluation Results

We compared the execution time of Condor with that of the state-of-the-art algorithm for planning - Helios [138], and also with the canonical planner implemented in Limes. We chose

[3] `http://www.linkedmdb.org/`

[4] The new data and a description of how they were constructed are available at `https://hobbitdata.informatik.uni-leipzig.de/LIGER/newDatasets/`

[5] Our complete experimental results can be found at `http://titan.informatik.uni-leipzig.de/kgeorgala/condor_results.zip`. Our open source code can be found at `http://limes.sf.net`

Limes because it is a state-of-the-art declarative framework for link discovery that ensures result completeness. Figure 8.5 shows the runtimes achieved by the different algorithm in different settings. As shown in Figures 8.5, 8.6, 8.7 and 8.8, Condor outperforms Canonical and Helios on all datasets. A Wilcoxon signed-rank test on the cumulative runtimes of the approaches (significance level = 99%) confirms that the differences in performance between Condor and the other approaches are statistically significant on all datasets. This observation and the statistical test clearly answer question Q_1: Condor outperforms the state of the art in planning by being able to generate more time-efficient plans than Helios and Canonical.

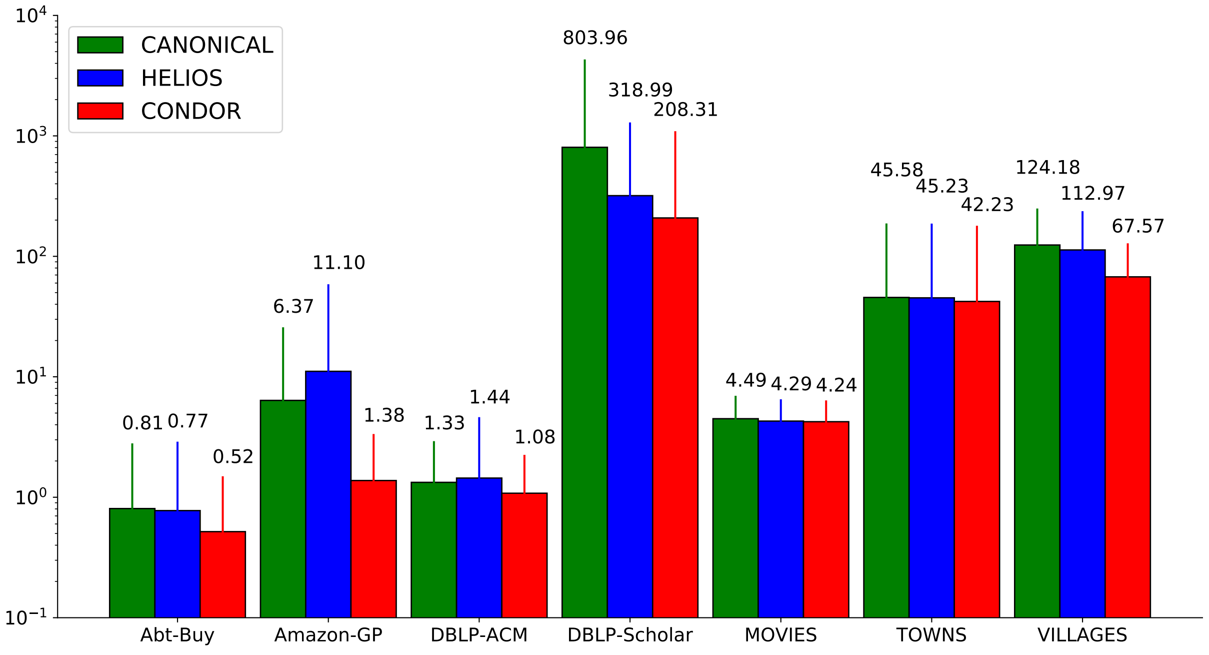
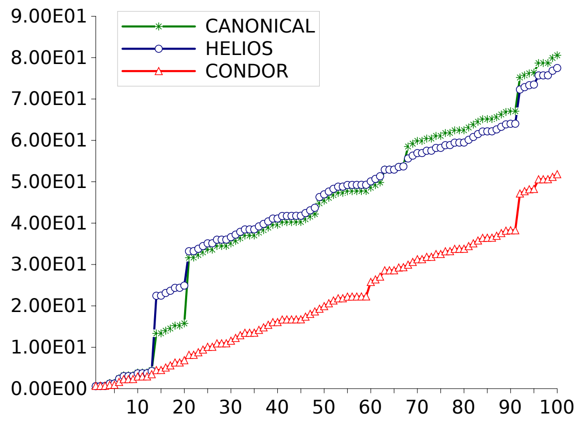


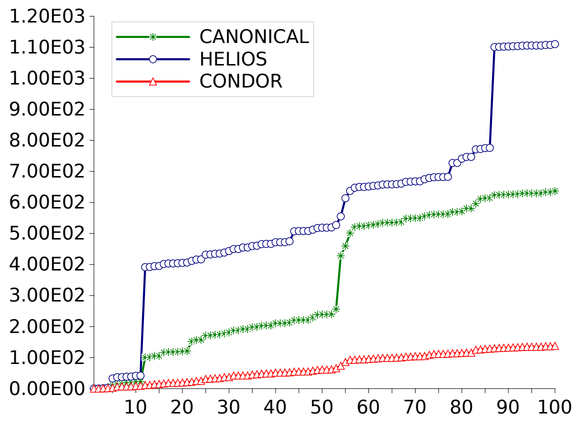
Figure 8.5: Mean and standard deviation of runtimes of Canonical, Helios and Condor for all LSs. The y-axis shows runtimes in seconds on a logarithmic scale. The numbers on top of the bars are the average runtimes.

Figure 8.5 shows that our approach performs best on Amazon-GP, where it can reduce the average runtime of the set of specifications by 78% compared to Canonical, making Condor 4.6 times faster. Moreover, for the same dataset, dynamic planning is 8.04 times more efficient than Helios. Note that finding a better plan than the canonical plan on this particular dataset is non-trivial (as shown by the Helios results). Here, our dynamic planning approach pays off by being able to revise the original and altering this plan at runtime early enough to achieve better results than both Canonical and Helios. The highest absolute difference is achieved on DBLP-Scholar, where Condor reduces the overall execution time of the Canonical planner on the 100 LSs by an average of approximately 600 s per specification. On the same dataset, the difference between Condor and Helios is approximately 110 s per LS.

The answer to our second question is that the benefits of the dynamic planning strategy are far superior to the time required by the re-planning scheme (as showed by Figure 8.5). Condor spends between 0.0005% (DBLP-Scholar) and 0.1% (Amazon-GP) of the overall runtime on planning. The specifications computed for the Amazon-GP dataset have, on average, the largest size in contrast to the other datasets. On this particular dataset, Condor spends less than 10 ms planning. We regard this result as particularly good, as using Condor brings larger benefits with growing specifications. Answer to Q_2: in our experiments, Condor invests less than 10 ms in planning.

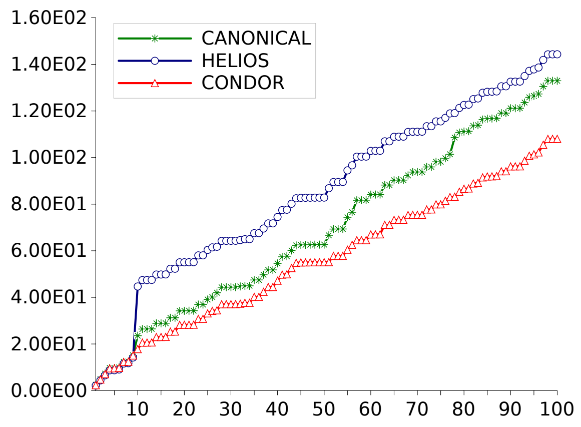


(a) Abt-Buy

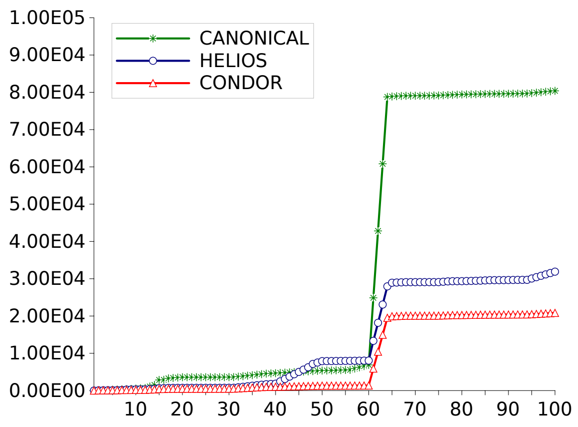


(b) Amazon-GP

Figure 8.6: Comparison of runtimes of Canonical, Helios and Condor on the Abt-Buy and Amazon-GP datasets of our evaluation data. The x-axis represents the number of specifications, the y-axis represents the cumulative execution times in seconds.



(a) DBLP-ACM

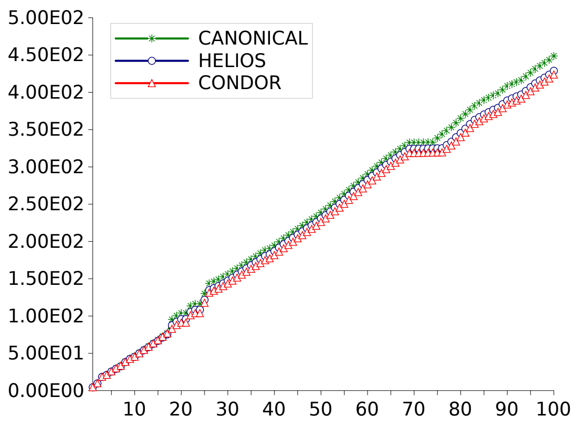


(b) DBLP-Scholar

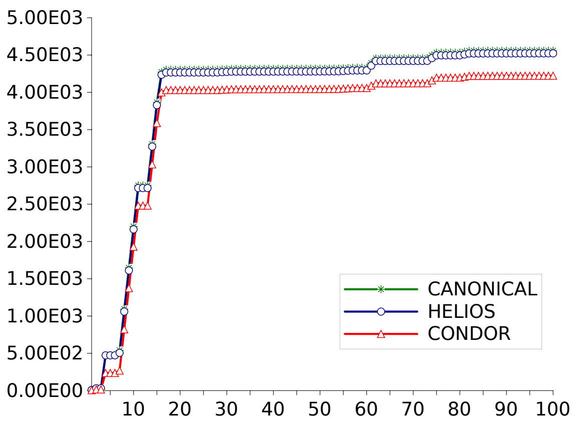
Figure 8.7: Comparison of runtimes of Canonical, Helios and Condor on the DBLP-ACM and DBLP-Scholar datasets of our evaluation data. The x-axis represents the number of specifications; the y-axis represents the cumulative execution times in seconds.

To answer Q_3, we also computed the runtime of LSs depending on their size (see Figures 8.9 and 8.10). For LSs of size 1, the execution times achieved by all three planners are most commonly comparable (difference of average runtimes = 0.02 s), since the plans produced are straight-forward and leave no room for improvement. For specifications of size 3, Condor is already capable of generating plans that are, on average, 7.5% faster than the canonical plans. The gap between Condor and the state of the art increases with the size of the specifications. For specifications of sizes 7 and more, Condor plans only necessitate 30.5%, resp. 55.7% of the time required by the plans generated by Canonical, resp. Helios.

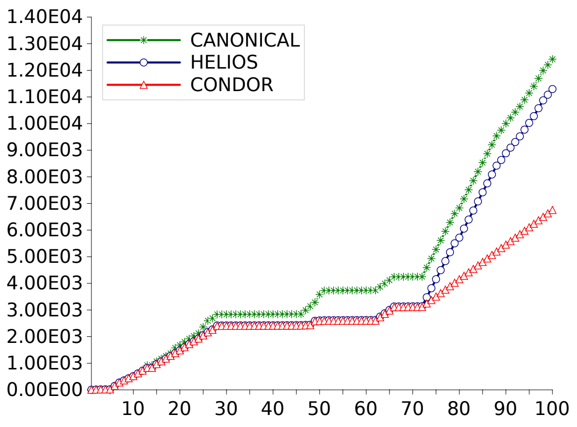
A careful study of the plan generated by Condor reveals that the re-use of previously executed portions of a LS and the use of subsumption are clearly beneficial to the execution runtime of large LSs. However, the study also shows that in a few cases, Condor creates a *filter-right* or *filter-left* plan where a *canonical* plan would have been better. This is due to some sub-optimal runtime approximations produced by the $re(P)$ function. Answer to Q_3: Condor's



(a) MOVIES



(b) TOWNS



(c) VILLAGES

Figure 8.8: Comparison of runtimes of Canonical, Helios and Condor on the last three datasets of our evaluation data. The x-axis represents the number of specifications; the y-axis represents the cumulative execution times in seconds.

performance gain over the state of the art grows with the size of the specifications.

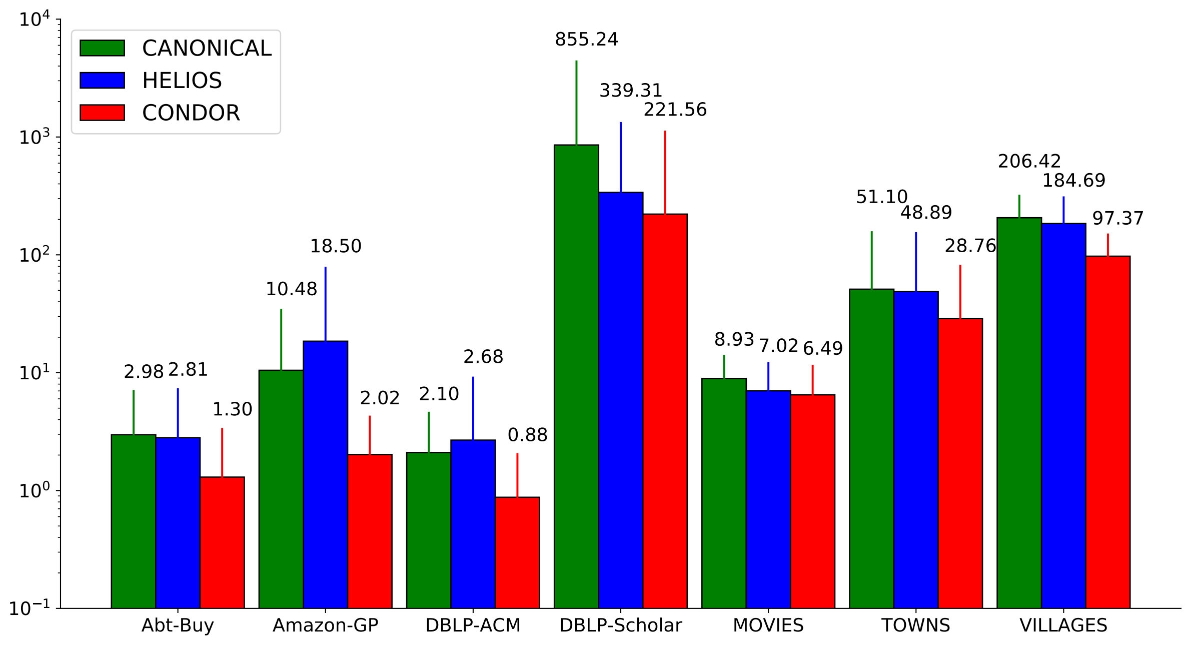


Figure 8.9: Mean and standard deviation of runtimes of Canonical, Helios and Condor for all LSs with size greater or equal to 3. The y-axis shows runtimes in seconds on a logarithmic scale. The numbers on top of the bars are the average runtimes.

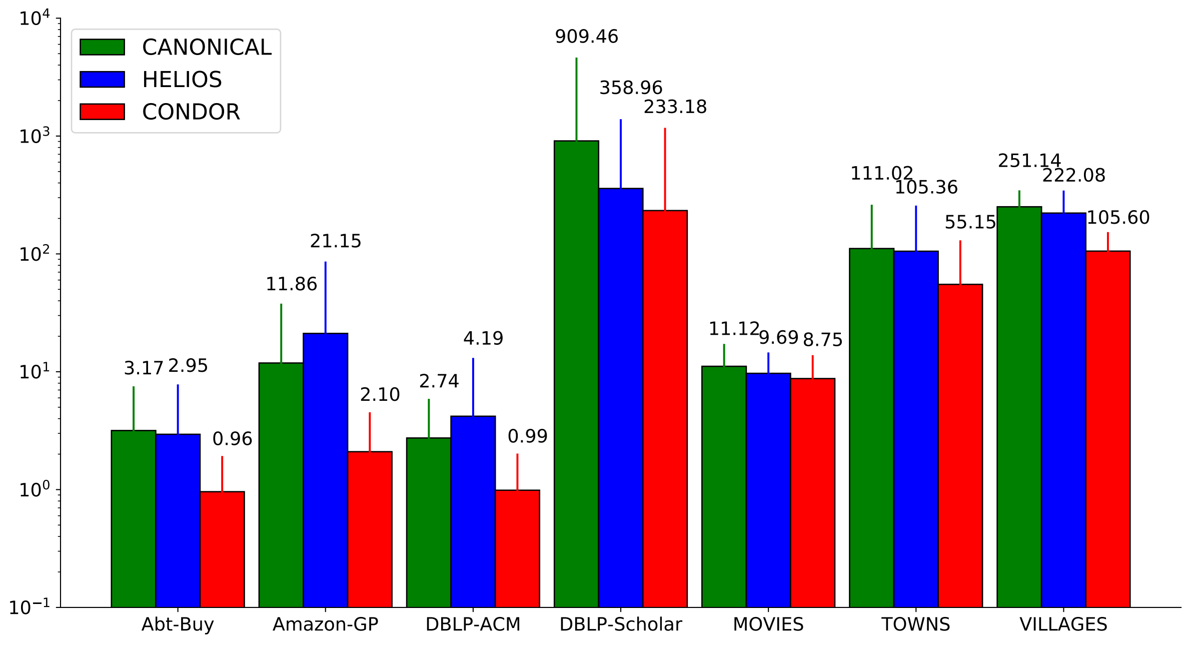


Figure 8.10: Mean and standard deviation of runtimes of Canonical, Helios and Condor for all LSs with size greater or equal to 5. The y-axis shows runtimes in seconds on a logarithmic scale. The numbers on top of the bars are the average runtimes.

9 Conclusions and Future Work

The goal of this thesis was to address the challenge of time-efficient LD by providing means to:

- improve the executing runtime of single measures, and
- accelerate the execution of whole specifications through planning, which demands the prediction of runtimes.

For the first aspect, we focused our research on two types of single measures: (1) temporal, and (2) string semantic similarities. In Chapter 4, we presented a time-efficient approach for the computation of temporal relations based on the reduction of Allen relations to 8 atomic relations that can be computed efficiently. We showed that by using simple sorting, we can reduce the complexity of computing any of these relations to $O(nlogn)$. Our experiments showed that our approach outperforms the state of the art, which is based on multidimensional blocking. In future work, we will extend the scalability of our approach by providing dedicated solutions for load balancing within a parallel execution setting. Moreover, we will study the incremental computation of temporal links on streams of data.

For the string semantic similarities, in Chapter 5, we presented hECATE, a generic framework for improving the runtime of edge-counting semantic similarities. Our evaluation of the framework shows that there is still a lot of potential in improving the runtime of semantic similarities for LD. We used hECATE to evaluate the performance of string similarities in LD on five datasets. Our evaluation shows that combining semantic similarities with string similarities can indeed increase the F-measure achieved by LD algorithms. This result is of central importance as it goes against current assumptions. The reason why we are indeed able to use semantic similarities for improving the F-measure of LD in some cases lies in the refinement operator employed by WOMBAT. In future works, we will investigate means that will allow improving the runtimes of semantic similarities, emphasize on how semantic similarities can handle homonyms, extend our works beyond edge-counting similarities and aim to classify datasets w.r.t. how suitable they are for semantic similarities.

Based on our findings for the two types of single measures, we performed a systematic survey on String Similarity Joins for Link Discovery, closing this research gap by presenting 54 SSJs that were published between 2008 and 2018. We divided our research scope into two main categories: (1) studies related to our systematic survey, and (2) published papers of SSJs for LD. For both categories, we stated the inclusion and exclusion criteria, and we followed a structured methodology of searching for corresponding publications, for which we presented in detail the different steps of our search along with numerical evidence. For the first category, we discussed our findings and showed the lack of systematic surveys regarding SSJs and their role in LD. For the second category, we presented our categorization criteria and the research questions we aimed to answer throughout our survey.

We divided the SSJs into two main categories: (1) filter-verify, and (2) tree-based approaches. The filter-verify approaches were further partitioned into 3 sub-categories based on signature generation technique they used. We showed the basic features of each SSJ such as the string similarities it targets, its signature scheme, the filtering/pruning techniques or tree structures used to minimize the set of candidate pairs, and any optimizations proposed for verification. In the evaluation section, we divided the SSJs based on the string similarity group they main to optimized (token-based, character-based or hybrid). For each group, we showed the basic trends followed over the past decade and how ideas involved to further optimize the efficiency of string similarities.

Throughout our systematic research, we identified three main challenges associated with SSJs—therewith answering $CS - Q_2$:

1. completeness vs. efficiency;

2. space vs. time complexity;

3. scalability.

The first challenge refers to the ability of each SSJ approach to maintain a low number of false positives/negatives by taking into consideration the computational costs required for high filtering/pruning power. For the filter-verify approaches, this challenge includes the decision of choosing the appropriate signature scheme. The current literature suggests that filter-verify approaches are not suitable for short string comparisons, since they are not able to select high-quality signatures [226, 117]. Choosing a single token as signature with low selectivity leads to mismatched pairs sharing the same tokens. As a result, the number of false positives increases. Regarding the tree-based approaches, using a trie structure for index leads to expensive traversals of complexity $O(B \times Z)$, where is B is the number of strings and Z is an average length of a string, which makes them inadequate for long strings. Additionally, in case of large Levenshtein distance thresholds, the produced tree structure becomes enormous with more active nodes that need further pruning. The Bi-Trie-PathStack method introduced in [59] and ***PreJoin*** [71] tried to solve the issue of tree structures for large edit-distance thresholds. Partition-based approaches are proven to be able to find good quality signatures for both short and long strings [117].

The second challenge involves the trade-off between having fast and time-efficient solutions for string similarities that come at the cost of space requirements, and vice versa. Some tree-based approaches overcome this issue by using of trie structures to index data strings that minimizes the index size, and perform string similarity joins efficiently without the cost of the verification stage. However, as explained before, inserting and searching nodes in a trie structure comes at the cost of efficiency. On the other hand, the filter-verify approaches need to store both the signatures and the original strings, since the verification step is performed in string level. Approaches such as ***Ed-Join*** [219], ***PPJoin*** [221], ***VChunkJoin*** [215] and ***MPJoin*** [162] tried to minimize the index space by proposing various bounds to minimize the size of signatures index. Furthermore, both categories of SSJs need to consider whether they keep the sets of signatures or indexes in memory for faster the storing/loading operations, or transfer everything to disk, risking performance deterioration. Longer signatures and larger indexes have a greater pruning power but lead to higher filtering time and space requirements, whereas shorter signatures and smaller indexes needs less space but make the comparisons slower.

The third challenge refers to the ability of each approach to deal with large amounts of data, while maintaining a low time complexity. Many SSJ approaches deal with this issue by implementing parallel approaches utilizing MapReduce, such as ***MPJoin*** [162] and ***PeARL*** [161]. However, parallel processing comes with two main considerations. First, data skewness among partitions might lead to pairs of strings to be processed more than once. Second, avoiding segmenting and replicating the data among partitions might not be a trivial issue. Therefore, many SSJ methods, such as ***SSJ-2*** [17], have focused on implementing additional techniques to overcome these problems by creating partition methodologies that consider the underlying data distribution.

The observations above led us to the following conclusions:

- The work that has been carried out in the field of SSJs for LD focuses on syntactic string similarities. Based on the evaluation results of Section 3.1.6, implementing and combining SSJs for both semantic and syntactic string similarities could be prove beneficial for the effectiveness of LD.
- Apart from providing theoretical guarantees for the runtime improvement of the edit-distance family, there has been limited experimental evaluation on any other metric of the aforementioned same family, apart from Levenhstein.
- Partition-based approaches provide a good solution for both short and long strings comparisons. Prefix-based approaches perform better in the presence of large strings, whereas tree-based approaches need further optimizations to deal with enormous tree structures.
- Parallel versions of SSJs are suitable for large datasets. Using MapReduce or Spark for small datasets creates an unnecessary overhead without improving the efficiency of SSJs significantly. Other distributed processing schemes remain an open research field.

For the second aspect, in Chapters 6, 7 and 8, we presented (1) a study of three different approximation functions that allow predicting the runtime of LSs, (2) the first dynamic planner for Link Discovery, Condor, and (3) the first (to the best of our knowledge) partial-recall LD approach. As we explained in Chapter 6, the selection of accurate runtime approximation models is a first necessary step towards the fast execution of LSs. The goal of Condor and Liger was to efficiently execute LSs, focusing on planning and LD under time constrains resp. Based on our findings from Section 6.2, we showed that on average, linear models are indeed the approach to chose to this end as they seem to overfit the least. Still, mixed models also perform in a satisfactory manner. Exponential models either fit very well or not at all and are thus not to be used. In future work, we will study further models for the evaluation of runtime and improve upon existing planning mechanisms for the declarative Link Discovery. In particular, we will consider other features when approximating runtimes, e.g., the distribution of characters in the strings to compare.

In Chapter 7, we provided a formal definition of a downward refinement operator with which we can detect subsumed LS with partial recall. We studied its characteristics and prove that our operator is finite, redundant, proper and incomplete. We used this operator to develop an algorithm for partial-recall Link Discovery. Additionally, we introduced an extension of said algorithm that takes into consideration the monotonicity of run times. Then, we evaluated our approach on 7 datasets derived from real data and showed that our approach scales to large datasets. Our results show that by using a downward refinement operator and insights pertaining to the subsumption of LS, we are able to detect a LS with guaranteed expected recall efficiently. Our extension of the original Liger algorithm with a monotonicity assumption pertaining to the run time of the LS was shown to be slightly better than the basic Liger implementation.

Also, we demonstrate that the results of partial-recall LD can be used to initialize supervised LD approaches without worsening their recall. In future work, we will build upon LIGER to guarantee the real selectivity and recall of our approaches with a given probability.

Based on the results from Section 8.2, we showed how our approach combines dynamic planning with subsumption and result caching to outperform the state of the art static planners by up to two orders of magnitude. A large number of questions were unveiled by our results. First, our results suggested that CONDOR 's runtimes can be improved further by improving the cost function underlying the approach. Hence, we will study the use of most complex regression approaches for approximating the runtime of metrics. Moreover, the parallel execution of plans will be studied in future.

Bibliography

[1] M. A. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine*, 14:491 – 498, 03 1995.

[2] S. Abiteboul, R. Hull, and V. Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[3] H. Abu Ahmad and H. Wang. An effective weighted rule-based method for entity resolution. *Distributed and Parallel Databases*, 36(3):593–612, Sep 2018.

[4] S. A. Aghili, D. Agrawal, and A. El Abbadi. Bft: Bit filtration technique for approximate string join in biological databases. In M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, editors, *String Processing and Information Retrieval*, pages 326–340, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[5] J. L. Aguirre, B. C. Grau, K. Eckert, J. Euzenat, A. Ferrara, R. W. Van Hague, L. Hollink, E. Jiménez-Ruiz, C. Meilicke, A. Nikolov, et al. Results of the ontology alignment evaluation initiative 2012. In *Proc. 7th ISWC workshop on ontology matching (OM)*, pages 73–115. No commercial editor., 2012.

[6] S. R. Alenazi and K. Ahmad. Record duplication detection in database: A review. *International Journal on Advanced Science, Engineering and Information Technology*, 6(6):838–845, 2016.

[7] K. F. Alfatmi and A. S. Vaidya. Survey of scalable string similarity joins. *International Journal of Computer Science and Information Technologies*, 6:194–197, 2015.

[8] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983.

[9] R. J. and Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, August 2003.

[10] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 635–644, New York, NY, USA, 2011. ACM.

[11] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 40–49, Washington, DC, USA, 2008. IEEE Computer Society.

[12] S. Auer, J. Lehmann, and A.-C. Ngonga Ngomo. *Introduction to Linked Data and Its Lifecycle on the Web*, pages 1–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[13] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM.

[14] A. Badarneh, A. Abdi, S. Shboul, and H. Najadat. Survey of similarity join algorithms based on mapreduce. *MATTER: International Journal of Science and Technology*, 2(1), 2016.

[15] L. Badea. Perfect Refinement Operators Can Be Flexible. In *Proceedings of the 14th European Conference on Artificial Intelligence*, ECAI'00, pages 266–270, Amsterdam, The Netherlands, The Netherlands, 2000. IOS Press.

[16] A. Banu, S. S. Fatima, and K. U. R. Khan. A Survey and Comparison of WordNet Based Semantic Similarity Measures. *International Journal of Computer Science And Technology*, 42(3):456–461, 2013.

[17] R. Baraglia, G. De Francisci Morales, and C. Lucchese. Document similarity self-join with mapreduce. In *2010 IEEE International Conference on Data Mining*, pages 731–736, Dec 2010.

[18] D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. Incremental Reasoning on Streams and Rich Background Knowledge. In *Proceedings of the 7th International Conference on The Semantic Web: Research and Applications - Volume Part I*, ESWC'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 131–140, New York, NY, USA, 2007. ACM.

[20] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. Rdf 1.1 turtle. *World Wide Web Consortium*, 2014.

[21] D. Beckett and B. McBride. Rdf/xml syntax specification (revised). *W3C recommendation*, 10(2.3), 2004.

[22] D. Ben-David, T. Domany, and A. Tarem. Enterprise data classification using semantic web technologies. In *International Semantic Web Conference*, pages 66–81. Springer, 2010.

[23] K. Bennett, M. C. Ferris, and Y. E. Ioannidis. A Genetic Algorithm for Database Query Optimization. In *Proceedings of the fourth International Conference on Genetic Algorithms*, pages 400–407, 1991.

[24] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[25] S. Bin, P. Westphal, J. Lehmann, and A. Ngonga. Implementing scalable structured machine learning for big data in the sake project. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1400–1407. IEEE, 2017.

[26] A. Bolles, M. Grawunder, and J. Jacobi. Streaming SPARQL - Extending SPARQL to Process Data Streams. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC'08, pages 448–462, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] A. Budanitsky and G. Hirst. Evaluating WordNet-based Measures of Lexical Semantic Relatedness. *Computational Linguistics*, 32(1):13–47, 2006.

[28] S. Chakrabarti, B. Dom, and P. Indyk. Enhanced hypertext categorization using hyperlinks. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 307–318, New York, NY, USA, 1998. ACM.

[29] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 34–43. ACM, 1998.

[30] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 5–, Washington, DC, USA, 2006. IEEE Computer Society.

[31] P. Christen. Probabilistic data generation for deduplication and data linkage. In *IDEAL*, 2005.

[32] P. Christen. Febrl - an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008.

[33] A. Cimmino and R. Corchuelo. A hybrid genetic-bootstrapping approach to link resources in the web of data. In F. J. de Cos Juez, J. R. Villar, E. A. de la Cal, Á. Herrero, H. Quintián, J. A. Sáez, and E. Corchado, editors, *Hybrid Artificial Intelligent Systems*, pages 145–157, Cham, 2018. Springer International Publishing.

[34] R. L. Cole and G. Graefe. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 150–160, New York, NY, USA, 1994. ACM.

[35] G. Costa, A. Cuzzocrea, G. Manco, and R. Ortale. *Data De-duplication: A Review*, pages 385–412. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[36] P. Courrieu. Fast computation of moore-penrose inverse matrices. *arXiv preprint arXiv:0804.4809*, 2008.

[37] V. Cross, P. Silwal, and D. Morell. Using a Reference Ontology with Semantic Similarity in Ontology Alignment. In *Proceedings of the 3rd ICBO*, 2012.

[38] I. F. Cruz, F. P. Antonelli, and C. Stroe. AgreementMaker: Efficient Matching for Large Real-World Schemas and Ontologies. *PVLDB*, 2:1586–1589, 2009.

[39] J. Cui, D. Meng, and Z.-T. Chen. Leveraging deletion neighborhoods and trie for efficient string similarity search and join. In A. Jaafar, N. Mohamad Ali, S. A. Mohd Noah, A. F. Smeaton, P. Bruza, Z. A. Bakar, N. Jamil, and T. M. T. Sembok, editors, *Information Retrieval Technology*, pages 1–13, Cham, 2014. Springer International Publishing.

[40] J. Cui, W. Wang, D. Meng, and Z. Liu. Continuous similarity join on data streams. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 552–559, Dec 2014.

[41] C. H. Dagli, N. Mehdiyev, J. Krumeich, D. Enke, D. Werth, and P. Loos. Complex Adaptive Systems San Jose, CA November 2-4, 2015 Determination of Rule Patterns in Complex Event Processing Using Machine Learning Techniques. *Procedia Computer Science*, 61:395 – 401, 2015.

[42] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, Mar. 1964.

[43] E. Daskalaki, G. Flouris, I. Fundulaki, and T. Saveta. Instance matching benchmarks in the era of linked data. *Journal of Web Semantics*, 39:1 – 14, 2016.

[44] J. de Freitas, G. Pappa, A. da Silva, M. Gonçalves, E. Moura, A. Veloso, A. Laender, and M. de Carvalho. Active learning genetic programming for record deduplication. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, July 2010.

[45] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[46] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *2014 IEEE 30th International Conference on Data Engineering*, pages 340–351, March 2014.

[47] Dhivyabharathi G V and S. Kumaresan. A survey on duplicate record detection in real world data. In *2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 01, pages 1–5, Jan 2016.

[48] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[49] C. F. Dorneles, R. Gonçalves, and R. dos Santos Mello. Approximate data instance matching: a survey. *Knowledge and Information Systems*, 27(1):1–21, Apr 2011.

[50] U. Draisbach and F. Naumann. A generalization of blocking and windowing algorithms for duplicate detection. In *Proceedings - 2011 International Conference on Data and Knowledge Engineering, ICDKE 2011*, pages 18 – 24, 10 2011.

[51] K. Dressler and A.-C. N. Ngomo. Time-efficient execution of bounded jaro-winkler distances. In *Proceedings of the 9th International Conference on Ontology Matching - Volume 1317*, OM'14, pages 37–48, Aachen, Germany, Germany, 2014. CEUR-WS.org.

[52] K. Dreßler and A.-C. Ngonga Ngomo. On the efficient execution of bounded jaro-winkler distances. *Semantic Web*, 8(2):185–196, 2017.

[53] V. Efthymiou, K. Stefanidis, and V. Christophides. Benchmarking blocking algorithms for web entities. *IEEE Transactions on Big Data*, pages 1–1, 2018.

[54] M. Ektefa, F. Sidi, H. Ibrahim, M. Jabar, and S. Memar. A comparative study in classification techniques for unsupervised record linkage model. *Journal of Computer Science*, 7:341–347, 01 2011.

[55] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, pages 265–268, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[56] J. Euzenat, A. Ferrara, C. Meilicke, A. Nikolov, J. Pane, F. Scharffe, P. Shvaiko, H. Stuckenschmidt, O. Šváb-Zazamal, V. Svátek, et al. Results of the ontology alignment evaluation initiative 2010. Technical report, University of Trento, 2011.

[57] N. Fanizzi, S. Ferilli, N. Di Mauro, and T. M. A. Basile. Spaces of Theories with Ideal Refinement Operators. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 527–532, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[58] C. Fellbaum. *WordNet: An Electronic Lexical Database.* Bradford Books, 1998.

[59] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *The VLDB Journal*, 21(4):437–461, Aug 2012.

[60] A. Ferrara, A. Nikolov, J. Noessner, and F. Scharffe. Evaluation of instance matching tools: The experience of oaei. *Journal of Web Semantics*, 21:49 – 60, 2013. Special Issue on Evaluation of Semantic Technologies.

[61] J. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12:175–179, 01 1984.

[62] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag. Set similarity joins on mapreduce: an experimental survey. *Proceedings of the VLDB Endowment*, 11:1110–1122, 06 2018.

[63] N. Gali, R. Mariescu-Istodor, and P. Fränti. Similarity measures for title matching. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 1548–1553, Dec 2016.

[64] K. Georgala, M. Hoffmann, and A.-C. N. Ngomo. An evaluation of models for runtime approximation in link discovery. In *Proceedings of the International Conference on Web Intelligence*, WI '17, pages 57–64, New York, NY, USA, 2017. ACM.

[65] K. Georgala, D. Obraczka, and A.-C. Ngonga Ngomo. Dynamic planning for link discovery. In A. Gangemi, R. Navigli, M.-E. Vidal, P. Hitzler, R. Troncy, L. Hollink, A. Tordai, and M. Alam, editors, *The Semantic Web*, pages 240–255, Cham, 2018. Springer International Publishing.

[66] K. Georgala, D. Obraczka, and A.-C. Ngonga Ngomo. Dynamic Planning for Link Discovery. In A. Gangemi, R. Navigli, M.-E. Vidal, P. Hitzler, R. Troncy, L. Hollink, A. Tordai, and M. Alam, editors, *The Semantic Web*, pages 240–255, Cham, 2018. Springer International Publishing.

[67] K. Georgala, M. Röder, M. A. Sherif, and A.-C. Ngonga Ngomo. Applying edge-counting semantic similarities to Link Discovery: Scalability and Accuracy. In *Proceedings of Ontology Matching Workshop 2020*, 2020.

[68] K. Georgala, M. A. Sherif, and A.-C. N. Ngomo. An efficient approach for the generation of allen relations. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, ECAI'16, pages 948–956, Amsterdam, The Netherlands, The Netherlands, 2016. IOS Press.

[69] K. Georgala, M. A. Sherif, and A.-C. Ngonga Ngomo. LIGER – Link Discovery with Partial Recall. In *Proceedings of Ontology Matching Workshop 2020*, 2020.

[70] M. Gollapalli, X. Li, I. Wood, and G. Governatori. Approximate record matching using hash grams. In *2011 IEEE 11th International Conference on Data Mining Workshops*, pages 504–511, Dec 2011.

[71] K. Gouda and M. Rashad. Efficient string edit similarity join algorithm. *Computing and Informatics*, 36(3):683–704, 2017.

[72] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[73] F. Grandi. Multi-temporal RDF Ontology Versioning. In *International Workshop on Ontology Dynamics*. CEUR-WS, 2009.

[74] F. Grandi. T-SPARQL: A TSQL2-like Temporal Query Language for RDF. In M. Ivanovic, B. Thalheim, B. Catania, and Z. Budimac, editors, *ADBIS (Local Proceedings)*, volume 639 of *CEUR Workshop Proceedings*, pages 21–30. CEUR-WS.org, 2010.

[75] 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 *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 491–500, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[76] T. Heath and C. Bizer. Linked data: Evolving the web into a global data space. *Synthesis lectures on the semantic web: theory and technology*, 1(1):1–136, 2011.

[77] J. Heflin and D. Song. Ontology instance linking: Towards interlinked knowledge graphs. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 4163–4169. AAAI Press, 2016.

[78] A. Hinze, D. M. Eyers, M. Hirzel, M. Weidlich, and S. Bhowmik, editors. *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS 2018, Hamilton, New Zealand, June 25-29, 2018*. ACM, 2018.

[79] P. Hitzler, M. Krtzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 1st edition, 2009.

[80] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench. In *Proceedings of ANZIIS '94*, pages 357–361, Nov 1994.

[81] Y. Huang, B. Niu, and C. Song. A partition-based bi-directional filtering method for string similarity joins. In X. L. Dong, X. Yu, J. Li, and Y. Sun, editors, *Web-Age Information Management*, pages 400–412, Cham, 2015. Springer International Publishing.

[82] J. Huber, T. Sztyler, J. Nößner, and C. Meilicke. Codi: Combinatorial optimization for data integration: results for oaei 2011. In *OM*, 2011.

[83] M. F. Huber, M. Voigt, and A.-C. N. Ngomo. Big data architecture for the semantic analysis of complex events in manufacturing. *Informatik 2016*, 2016.

[84] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 103–114, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[85] R. Isele and C. Bizer. Active Learning of Expressive Linkage Rules using Genetic Programming. *Web Semantics: Science, Services and Agents on the World Wide Web*, 23(0), 2013.

[86] R. Isele, A. Jentzsch, and C. Bizer. Efficient Multidimensional Blocking for Link Discovery without losing Recall. In *WebDB*, 2011.

[87] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 299–310, New York, NY, USA, 1999. ACM.

[88] P. Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[89] L. Jia. A survey on set similarity search and join. *International Journal of Performability Engineering*, 14, 02 2018.

[90] Y. Jiang, D. Deng, J. Wang, G. Li, and J. Feng. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 341–348, New York, NY, USA, 2013. ACM.

[91] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *Proc. VLDB Endow.*, 7(8):625–636, Apr. 2014.

[92] J. Jupin, J. Y. Shi, and E. C. Dragut. Psh: A probabilistic signature hash method with hash neighborhood candidate generation for fast edit-distance string comparison on big data. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 122–127, Dec 2016.

[93] C.-C. Kanne and G. Moerkotte. Histograms reloaded: The merits of bucket diversity. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 663–674. ACM, 2010.

[94] A. Karampelas and G. A. Vouros. Time and space efficient large-scale link discovery using string similarities. In *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, WIMS '18, pages 26:1–26:9, New York, NY, USA, 2018. ACM.

[95] M. Kazimianec and N. Augsten. Pg-join: Proximity graph based string similarity joins. In J. Bayard Cushing, J. French, and S. Bowers, editors, *Scientific and Statistical Database Management*, pages 274–292, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[96] M. Kazimianec and N. Augsten. Pg-skip: Proximity graph based clustering of long strings. In J. X. Yu, M. H. Kim, and R. Unland, editors, *Database Systems for Advanced Applications*, pages 31–46, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[97] M. Kejriwal and D. P. Miranker. Semi-supervised instance matching using boosted classifiers. In F. Gandon, M. Sabou, H. Sack, C. d'Amato, P. Cudré-Mauroux, and A. Zimmermann, editors, *The Semantic Web. Latest Advances and New Domains*, pages 388–402, Cham, 2015. Springer International Publishing.

[98] B. Kitchenham. Procedures for Performing Systematic Reviews. Technical report tr/se-0401, Keele University, Department of Computer Science, Keele University, UK, 2004.

[99] G. Klyne. Resource description framework (rdf): Concepts and abstract syntax. *http://www. w3. org/TR/2004/REC-rdf-concepts-20040210/*, 2004.

[100] D. Knuth. *The Art of Computer Programming.* Addison-Wesley, United States, 1968.

[101] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient deduplication with hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012.

[102] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.

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

[104] H. Kopcke, A. Thor, and E. Rahm. Learning-based approaches for matching web data entities. *IEEE Internet Computing*, 14(4):23–31, July 2010.

[105] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3:484–493, 09 2010.

[106] E. F. Krause. *Taxicab geometry: An adventure in non-Euclidean geometry.* Courier Corporation, 1986.

[107] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[108] D. Le-Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 370–388, Berlin, Heidelberg, 2011. Springer-Verlag.

[109] C. Leacock and M. Chodorow. Combining local context and wordnet similarity for word sense identification. In *WordNet: An Electronic Lexical Database*, volume 49, pages 265–, 01 1998.

[110] S. Lee, J. Lee, and S.-w. Hwang. Scalable entity matching computation with materialization. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2353–2356, New York, NY, USA, 2011. ACM.

[111] J. Lehmann and C. Haase. Ideal Downward Refinement in the EL Description Logic. In *Proceedings of the 19th International Conference on Inductive Logic Programming*, ILP'09, pages 73–87, Berlin, Heidelberg, 2010. Springer-Verlag.

[112] J. Lehmann and P. Hitzler. Foundations of Refinement Operators for Description Logics. In H. Blockeel, J. Ramon, J. W. Shavlik, and P. Tadepalli, editors, *Inductive Logic Programming, 17th International Conference, ILP 2007, Corvallis, OR, USA, June 19-21, 2007*, volume 4894 of *Lecture Notes in Computer Science*, pages 161–174. Springer, 2007. Best Student Paper Award.

[113] J. Lehmann and P. Hitzler. Concept Learning in Description Logics Using Refinement Operators. *Machine Learning journal*, 78(1-2):203–250, 2010.

[114] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, Feb. 1966.

[115] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *2008 IEEE 24th International Conference on Data Engineering*, pages 257–266, April 2008.

[116] G. Li, D. Deng, and J. Feng. A partition-based method for string similarity joins with edit-distance constraints. *ACM Trans. Database Syst.*, 38(2):9:1–9:33, July 2013.

[117] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, Nov. 2011.

[118] Y. Li, Z. A. Bandar, and D. McLean. An approach for measuring semantic similarity between words using multiple information sources. *IEEE Trans. on Knowl. and Data Eng.*, 15(4):871–882, July 2003.

[119] C. Lin, H. Yu, W. Weng, and X. He. Large-scale similarity join with edit-distance constraints. In S. S. Bhowmick, C. E. Dyreson, C. S. Jensen, M. L. Lee, A. Muliantara, and B. Thalheim, editors, *Database Systems for Advanced Applications*, pages 328–342, Cham, 2014. Springer International Publishing.

[120] M. Loskyll, J. Schlick, S. Hodek, L. Ollinger, T. Gerber, and B. Pîrvu. Semantic service discovery and orchestration for manufacturing processes. In *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8, Sept 2011.

[121] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 373–384, New York, NY, USA, 2013. ACM.

[122] Q. Lu and L. Getoor. Link-based classification. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 496–503, 2003.

[123] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *Proc. VLDB Endow.*, 9(9):636–647, May 2016.

[124] A. Mazeika and M. H. Böhlen. Cleansing databases of misspelled proper nouns. In *CleanDB*, 2006.

[125] J. P. McCrae and P. Buitelaar. Linking Datasets Using Semantic Textual Similarity. *CYBERNETICS AND INFORMATION TECHNOLOGIES*, 18(1):109–123, 2018.

[126] D. G. Mestre, C. E. Pires, and D. C. Nascimento. Adaptive sorted neighborhood blocking for entity matching with mapreduce. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 981–987, New York, NY, USA, 2015. ACM.

[127] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow.*, 5(8):704–715, Apr. 2012.

[128] D. Moher, A. Liberati, J. Tetzlaff, D. G. Altman, and T. P. Group. Preferred Reporting Items for Systematic Reviews and Meta-Analyses: The PRISMA Statement. *PLOS Medicine*, 6(7):1–6, 07 2009.

[129] A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of the SIGMOD 1997 Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 23–29, Tuscon, AZ, May 1997.

[130] J. J. Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[131] F. Nafis and D. Chiadmi. Methods and systems for the linked data. In A. El Oualkadi, F. Choubani, and A. El Moussati, editors, *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*, pages 587–592, Cham, 2016. Springer International Publishing.

[132] K. Narita, S. Nakadai, and T. Araki. Landmark-join: Hash-join based string similarity joins with edit distance constraints. In A. Cuzzocrea and U. Dayal, editors, *Data Warehousing and Knowledge Discovery*, pages 180–191, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[133] M. Nashaat, A. Ghosh, J. Miller, S. Quader, C. Marston, and J.-F. Puget. Hybridization of active learning and data programming for labeling large industrial datasets. *2018 IEEE International Conference on Big Data (Big Data)*, pages 46–55, 2018.

[134] M. Nentwig, M. Hartung, A.-C. Ngonga Ngomo, and E. Rahm. A survey of current link discovery frameworks. *Semantic Web*, 8(3):419–436, 2017.

[135] A.-C. N. Ngomo and S. Auer. Limes: A time-efficient approach for large-scale link discovery on the web of data. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pages 2312–2317. AAAI Press, 2011.

[136] A.-C. N. Ngomo and K. Lyko. Unsupervised learning of link specifications: deterministic vs. non-deterministic. In *OM*, 2013.

[137] A.-C. Ngonga Ngomo. On Link Discovery using a Hybrid Approach. *Journal on Data Semantics*, 1(4):203–217, 2012.

[138] A.-C. Ngonga Ngomo. Helios – execution optimization for link discovery. In *The Semantic Web – ISWC 2014*, pages 17–32, Cham, 2014. Springer International Publishing.

[139] A.-C. Ngonga Ngomo and S. Auer. LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In *Proceedings of IJCAI*, 2011.

[140] A.-C. Ngonga Ngomo, J. Lehmann, S. Auer, and K. Höffner. RAVEN – Active Learning of Link Specifications. In *Proceedings of OM@ISWC*, 2011.

[141] A.-C. Ngonga Ngomo and K. Lyko. *EAGLE: Efficient Active Learning of Link Specifications Using Genetic Programming*, pages 149–163. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[142] A.-C. Ngonga Ngomo and K. Lyko. Unsupervised learning of link specifications: deterministic vs. non-deterministic. In *Proceedings of the Ontology Matching Workshop*, 2013.

[143] A.-C. Ngonga Ngomo, K. Lyko, and V. Christen. Coala—correlation-aware active learning of link specifications. In *Proceedings of ESWC*, 2013.

[144] S.-H. Nienhuys-Cheng, P. R. J. van der Laag, and L. W. N. van der Torre. *Constructing refinement operators by decomposing logical implication*, pages 178–189. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[145] A. Nikolov, M. d'Aquin, and E. Motta. Unsupervised learning of link discovery configuration. In *9th Extended Semantic Web Conference (ESWC 2012)*, 2012.

[146] A. Nikolov, A. Ferrara, and F. Scharffe. Data linking for the semantic web. *Int. J. Semant. Web Inf. Syst.*, 7(3):46–76, July 2011.

[147] A. Nikolov, V. Uren, and E. Motta. Knofuss: A comprehensive architecture for knowledge fusion. In *Proceedings of the 4th International Conference on Knowledge Capture*, K-CAP '07, pages 185–186, New York, NY, USA, 2007. ACM.

[148] X. Niu, S. Rong, H. Wang, and Y. Yu. An Effective Rule Miner for Instance Matching in a Web of Data. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 1085–1094, New York, NY, USA, 2012. ACM.

[149] X. Niu, S. Rong, Y. Zhang, and H. Wang. Zhishi.links results for oaei 2011. In *Proceedings of the 6th International Conference on Ontology Matching - Volume 814*, OM'11, pages 220–227, Aachen, Germany, Germany, 2011. CEUR-WS.org.

[150] X. Niu, S. Rong, Y. Zhang, and H. Wang. Zhishi.links results for OAEI 2011. *Ontology Matching*, page 220, 2011.

[151] D. Obraczka and A.-C. N. Ngomo. Dragon: Decision tree learning for link discovery. In *19TH International Conference On Web Engineering*. Springer, 2019.

[152] G. Papadakis, G. Papastefanatos, T. Palpanas, and M. Koubarakis. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking. In *EDBT*, pages 221–232, 2016.

[153] U. Pfeifer, T. Poersch, and N. Fuhr. Retrieval effectiveness of proper name search methods. *Information Processing and Management*, 32(6):667–679, 1996.

[154] D. D. Prasetya, A. P. Wibawa, and T. Hirashima. The performance of text similarity algorithms. *International Journal of Advances in Intelligent Informatics*, 4(1):63–69, 2018.

[155] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1033–1044, New York, NY, USA, 2011. ACM.

[156] R. Rada, H. Mili, E. Bicknell, and M. Blettner. Development and application of a metric on semantic nets. *IEEE Trans. Systems, Man, and Cybernetics*, 19:17–30, 1989.

[157] E. Rajabi and S.-M.-R. Beheshti. *Interlinking Big Data to Web of Data*, pages 133–145. Springer International Publishing, Cham, 2016.

[158] E. Rajabi, M. Sicilia, and S. Sanchez-Alonso. An empirical study on the evaluation of interlinking tools on the web of data. *Journal of Information Science*, 40, 06 2014.

[159] B. Ramadan and P. Christen. Unsupervised blocking key selection for real-time entity resolution. In T. Cao, E.-P. Lim, Z.-H. Zhou, T.-B. Ho, D. Cheung, and H. Motoda, editors, *Advances in Knowledge Discovery and Data Mining*, pages 574–585, Cham, 2015. Springer International Publishing.

[160] G. Recchia and M. M. Louwerse. A comparison of string similarity measures for toponym matching. In *Proceedings of The First ACM SIGSPATIAL International Workshop on Computational Models of Place*, COMP '13, pages 54:54–54:61, New York, NY, USA, 2013. ACM.

[161] A. Rheinländer and U. Leser. Scalable sequence similarity search and join in main memory on multi-cores. In M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Traff, G. Vallée, and J. Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, pages 13–22, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

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

[163] L. Richards, L. Antonie, S. Areibi, G. Grewal, K. Inwood, and J. A. Ross. Comparing classifiers in historical census linkage. In *2014 IEEE International Conference on Data Mining Workshop*, pages 1086–1094, Dec 2014.

[164] M. Rinne, E. Blomqvist, R. Keskisärkkä, and E. Nuutila. Event processing in RDF. In *Proceedings of the 4th International Conference on Ontology and Semantic Web Patterns-Volume 1188*, pages 52–64. CEUR-WS. org, 2013.

[165] M. Rinne, E. Nuutila, and S. Törmä. INSTANS: High-Performance Event Processing with Standard RDF and SPARQL. In *Proceedings of the International Semantic Web Conference (ISWC) 2012 Posters & Demonstrations Track, Boston, USA, November 11-15, 2012*, 2012.

[166] M. A. Rodríguez and M. J. Egenhofer. Determining semantic similarity among entity classes from different ontologies. *IEEE transactions on knowledge and data engineering*, 15(2):442–456, 2003.

[167] 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 *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1059–1070, April 2017.

[168] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. H. Tung. Efficient and scalable processing of string similarity join. *IEEE Trans. on Knowl. and Data Eng.*, 25(10):2217–2230, Oct. 2013.

[169] C. Rong, Y. N. Silva, and C. Li. String similarity join with different similarity thresholds based on novel indexing techniques. *Frontiers of Computer Science*, 11(2):307–319, Apr 2017.

[170] A. Rosenfeld, R. A. Hummel, and S. W. Zucker. Scene labeling by relaxation operations. *IEEE Transactions on Systems, Man, and Cybernetics*, 6:420–433, 1976.

[171] A. Saeedi, E. Peukert, and E. Rahm. Comparative evaluation of distributed clustering schemes for multi-source entity resolution. In M. Kirikova, K. Nørvåg, and G. A. Papadopoulos, editors, *Advances in Databases and Information Systems*, pages 278–293, Cham, 2017. Springer International Publishing.

[172] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A.-C. N. Ngomo. *LSQ: The Linked SPARQL Queries Dataset*, pages 261–269. Springer International Publishing, Cham, 2015.

[173] M. Saleem and A.-C. Ngonga Ngomo. HiBISCuS: Hypergraph-based source selection for SPARQL endpoint federation. In *Extended Semantic Web Conference (ESWC 2014)*, 2014.

[174] S. Sami and L. George. A comparative study for string metrics and the feasibility of joining them as combined text similarity measures. *The Scientific Journal of Koya University (ARO)*, 5:6–18, 10 2017.

[175] A. Samiei and F. Naumann. Cluster-based sorted neighborhood for efficient duplicate detection. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 202–209, Dec 2016.

[176] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 743–754, New York, NY, USA, 2004. ACM.

[177] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. In *Proceedings of International Conference on Very Large Databases (VLDB)*, September 2014.

[178] T. Saveta, E. Daskalaki, G. Flouris, I. Fundulaki, M. Herschel, and A.-C. N. Ngomo. Lance: Piercing to the heart of instance matching tools. In M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, K. Thirunarayan, and S. Staab, editors, *The Semantic Web - ISWC 2015*, pages 375–391, Cham, 2015. Springer International Publishing.

[179] R. Schnell, T. Bachteler, and J. Reiher. Privacy-preserving record linkage using bloom filters. *BMC Medical Informatics and Decision Making*, 9(1):41, Aug 2009.

[180] Z. Sehili, L. Kolb, C. Borgs, R. Schnell, and E. Rahm. Privacy preserving record linkage with ppjoin. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 85–104, 2015.

[181] S. E. Seker, O. Altun, U. Ayan, and C. Mert. A novel string distance function based on most frequent k characters. *International Journal of Machine Learning and Computing*, 4(2), 2014.

[182] P. Selvaramalakshmi, S. H. Ganesh, and J. J. Manoharan. Survey of string similarity join algorithms on large scale data. *Int. J. Innov. Eng. Technol.(IJIET)*, pages 100–104, 2016.

[183] P. Selvaramalakshmi, S. H. Ganesh, and F. Tushabe. A novel ssps framework for string similarity join. *International Journal of Computer Applications*, 160(1):32–38, Feb 2017.

[184] H. Shang and T. h. Merrettal. Tries for approximate string matching. *IEEE Trans. on Knowl. and Data Eng.*, 8(4):540–547, Aug. 1996.

[185] E. Y. Shapiro. Inductive inference of theories from facts. In J. L. Lassez and G. D. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 199–255. The MIT Press, 1991.

[186] S. Shekarpour, S. Auer, A.-C. Ngonga Ngomo, D. Gerber, S. Hellmann, and C. Stadler. Keyword-driven sparql query generation leveraging background knowledge. In *International Conference on Web Intelligence*, 2011.

[187] S. Shekarpour, A.-C. Ngonga Ngomo, and S. Auer. Question answering on interlinked data. In *Proceedings of the 22nd international conference on World Wide Web*, pages 1145–1156. ACM, 2013.

[188] M. Sherif, A.-C. Ngonga Ngomo, and J. Lehmann. WOMBAT - A Generalization Approach for Automatic Link Discovery. In *14th Extended Semantic Web Conference, Portorož, Slovenia, 28th May - 1st June 2017*. Springer, 2017.

[189] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.

[190] Y. N. Silva, J. Reed, K. Brown, A. Wadsworth, and C. Rong. An experimental survey of mapreduce-based similarity joins. In L. Amsaleg, M. E. Houle, and E. Schubert, editors, *Similarity Search and Applications*, pages 181–195, Cham, 2016. Springer International Publishing.

[191] P. Smeros and M. Koubarakis. Discovering Spatial and Temporal Links among RDF Data. In *Proceedings of the 25th World Wide Web Conference Workshop*, 2016.

[192] M. D. Soo and R. T. Snodgrass. Temporal Data Types. In *The TSQL2 Temporal Query Language*, pages 119–148. Springer, 1995.

[193] T. Soru, E. Marx, and A.-C. Ngonga Ngomo. ROCKER – a refinement operator for key discovery. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015*, 2015.

[194] T. Soru and A.-C. N. Ngomo. Rapid execution of weighted edit distances. In *Proceedings of the Ontology Matching Workshop*, 2013.

[195] T. Soru and A.-C. N. Ngomo. A comparison of supervised learning classifiers for link discovery. In *Proceedings of the 10th International Conference on Semantic Systems*, pages 41–44, New York, NY, USA, 2014. ACM.

[196] T. Soru and A.-C. Ngonga Ngomo. A comparison of supervised learning classifiers for link discovery. In *SEMANTICS*, volume 2014, 09 2014.

[197] K. Stefanidis, V. Christophides, and V. Efthymiou. Web-scale blocking, iterative and progressive entity resolution. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1459–1462, April 2017.

[198] R. C. Steorts, S. L. Ventura, M. Sadinle, and S. E. Fienberg. A comparison of blocking methods for record linkage. In J. Domingo-Ferrer, editor, *Privacy in Statistical Databases*, pages 253–268, Cham, 2014. Springer International Publishing.

[199] D. Sun and X. Wang. Mls-join: An efficient mapreduce-based algorithm for string similarity self-joins with edit distance constraint. In *ICCCS*, 2018.

[200] J. Tappolet and A. Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, pages 308–322, Berlin, Heidelberg, 2009. Springer-Verlag.

[201] R. Tous and J. Delgado. A vector space model for semantic similarity calculation and OWL ontology alignment. In *International Conference on Database and Expert Systems Applications*, pages 307–316. Springer, 2006.

[202] I. Trummer and C. Koch. Multi-objective Parametric Query Optimization. *Proc. VLDB Endow.*, 8(3):221–232, Nov. 2014.

[203] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1):100 – 118, 1985. International Conference on Foundations of Computation Theory.

[204] C. Unger, L. Bühmann, J. Lehmann, A.-C. N. Ngomo, D. Gerber, and P. Cimiano. Template-based Question Answering over RDF data. In *Proceedings of the 21st international conference on World Wide Web*, pages 639–648, 2012.

[205] C. Unger, C. Forascu, V. Lopez, A.-C. Ngonga Ngomo, E. Cabrio, P. Cimiano, and S. Walter. Question Answering over Linked Data (QALD-4). In L. Cappellato, N. Ferro, M. Halvey, and W. Kraaij, editors, *Working Notes for CLEF 2014 Conference*, Sheffield, United Kingdom, Sept. 2014.

[206] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based Query Scrambling for Initial Delays. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 130–141, New York, NY, USA, 1998. ACM.

[207] A. Valdestilhas, T. Soru, and A.-C. Ngonga Ngomo. A high-performance approach to string similarity using most frequent k characters. In *Ontology Matching Workshop (co-located with ISWC 2017)*, 10 2017.

[208] P. R. J. van der Laag and S.-H. Nienhuys-Cheng. *Existence and nonexistence of complete refinement operators*, pages 307–322. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[209] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 495–506, New York, NY, USA, 2010. ACM.

[210] J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. Discovering and maintaining links on the web of data. In *International Semantic Web Conference*, 2009.

[211] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string metrics for matching names and records. *Proc of the KDD Workshop on Data Cleaning and Object Consolidation*, 10 2003.

[212] S. Wandelt, D. Deng, S. Gerdjikov, S. Mishra, P. Mitankin, M. Patil, E. Siragusa, A. Tiskin, W. Wang, J. Wang, and U. Leser. State-of-the-art in string similarity search and join. *SIGMOD Rec.*, 43(1):64–76, May 2014.

[213] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. *Data Engineering (ICDE)*, pages 458–469, 04 2011.

[214] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 85–96, New York, NY, USA, 2012. ACM.

[215] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Transactions on Knowledge and Data Engineering*, 25(8):1916–1929, Aug 2013.

[216] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 759–770, New York, NY, USA, 2009. ACM.

[217] W. E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, pages 354–359, 1990.

[218] Z. Wu and M. Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32Nd Annual Meeting on Association for Computational Linguistics*, ACL '94, pages 133–138, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.

[219] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endow.*, 1(1):933–944, Aug. 2008.

[220] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 131–140, New York, NY, USA, 2008. ACM.

[221] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.*, 36(3):15:1–15:41, Aug. 2011.

[222] C. Xiao, Y. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1:933–944, 08 2008.

[223] C. Yan, X. Zhao, Q. Zhang, and Y. Huang. Efficient string similarity join in multi-core and distributed systems. *PLOS ONE*, 12(3):1–16, 03 2017.

[224] S. Yan, D. Lee, M.-Y. Kan, and C. Lee Giles. Adaptive sorted neighborhood methods for efficient record linkage. *Proceedings of the ACM International Conference on Digital Libraries*, pages 185–194, 01 2007.

[225] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417, Jun 2016.

[226] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 915–926, New York, NY, USA, 2010. ACM.

[227] J. Zhu, X. Wu, X. Lin, C. Huang, G. P. C. Fung, and Y. Tang. A novel multiple layers name disambiguation framework for digital libraries using dynamic clustering. *Scientometrics*, 114(3):781–794, Mar 2018.

.1 Annex

The following equations define some well-known string similarities between two strings r and g. $tokens(r)$ and $tokens(g)$ represent the tokens sets of r and g resp.

$$\text{overlap: } sim_{overlap}(r,g) = |tokens(r) \cap tokens(g)|$$

$$\text{jaccard: } sim_{jaccard}(r,g) = \frac{|tokens(r) \cap tokens(g)|}{|tokens(r) \cup tokens(g)|}$$

$$\text{cosine: } sim_{cosine}(r,g) = \frac{|tokens(r) \cap tokens(g)|}{\sqrt{|tokens(r)| * |tokens(g)|}}$$

$$\text{dice: } sim_{dice}(r,g) = \frac{2|tokens(r) \cap tokens(g)|}{|tokens(r)| + |tokens(g)|}$$

The Levenshtein distance ($distance_{Levenshtein}$) between two strings r and g is the minimum number of single-character edits (insertions, deletions or substitutions) required to change g into r. Using dynamic programming, we calculate the distance as follows: Given two strings r and g, we define a matrix V with $|r|+1$ rows and $|g|+1$ columns that is used to compute their edit distance, where $|r|$ and $|g|$ is the length of r and g resp. as the number of characters included within each string. A cell $V[i][j]$ is the Levenshtein distance between the prefix of r with length i and the prefix of g with length j. Initially, we set $V[i][0] = 0$ and $V[0][j] = 0$ for $0 \leq i \leq |r|$ and $0 \leq j \leq |g|$. Each value of the matrix is computed using the following equation:

$$V[i][j] = min(V[i-1][j] + 1, V[i][j-1] + 1, M[i-1][j-1] + z)$$

where $z = 0$ if the $i-$th character of r is equal to the $j-$th character of g, and $z = 1$ otherwise. Once each value of the matrix is computed, the Levenshtein distance between r and g is:

$$distance_{Levenshtein}(r,g) = V[|r|+1][|g|+1]$$

The normalized Levenshtein distance is computed as follows:

$$sim_{NormLevenshtein}(r,g) = \frac{distance_{Levenshtein}(r,g)}{max(|r|,|g|)}$$

The Levenshtein similarity is be computed as follows:

$$sim_{Levenshtein}(r,g) = 1 - \frac{distance_{Levenshtein}(r,g)}{max(|r|,|g|)}$$

The Hamming distance between two strings r and r' of *equal length* is the number of positions at which the corresponding characters are different. Assuming that V_r and $V_{r'}$ are two arrays of equal length $|r| = |r'|$, each cell $V_r[i]$ and $V_{r'}[j]$ ($0 \leq i \leq |r|$, $0 \leq j \leq |r'|$) corresponds to the character at position i and j of r and r' resp., the Hamming distance ($distance_{Hamming}$) is calculated as follows:

$$sim_{Hamming}(r,r') = \sum_{i,j=0}^{|r|} 1_{V_r[i] \neq V_{r'}[j]}$$

where $1_{V_r[i] \neq V_{r'}[j]}$ is the indicator function equal to 1 when $V_r[i] \neq V_{r'}[j]$ and equal to 0 otherwise.

.2 Publications

- **Scalable Link Discovery for Modern Data-Driven Applications** by Kleanthi Georgala in Proceedings of the The 15th International Semantic Web Conference (ISWC2016) 2016, Doctoral Consortium Track, Kobe, Japan, 17. October - 21. October 2016
- **An Efficient Approach for the Generation of Allen Relations** by Kleanthi Georgala, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo in Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI) 2016, The Hague, 29. August - 02. September 2016
- **MOCHA2017: The Mighty Storage Challenge at ESWC** 2017 by Kleanthi Georgala, Mirko Spasić, Milos Jovanovik, Henning Petzka, Michael Röder, and Axel-Cyrille Ngonga Ngomo in Semantic Web Evaluation Challenge
- **An Evaluation of Models for Runtime Approximation in Link Discovery** by Kleanthi Georgala, Michael Hoffmann, and Axel-Cyrille Ngonga Ngomo in Proceedings of the International Conference on Web Intelligence, 2017
- **MOCHA2018: The Mighty Storage Challenge at ESWC** 2018 by Kleanthi Georgala, Mirko Spasić, Milos Jovanovik, Vassilis Papakonstantinou, Claus Stadler, Michael Röder, and Axel-Cyrille Ngonga Ngomo in Semantic Web Evaluation Challenges
- **Dynamic Planning for Link Discovery** by Kleanthi Georgala, Daniel Obraczka, and Axel-Cyrille Ngonga Ngomo in The Semantic Web, ESWC 2018, Lecture Notes in Computer Science
- **Applying edge-counting semantic similarities to Link Discovery: Scalability and Accuracy** by Kleanthi Georgala, Mohamed Ahmed Sherif, Michael Röder and Axel-Cyrille Ngonga Ngomo in Proceedings of the 15th International Workshop on Ontology Matching 2020 (OM-2020), collocated with the 19th International Semantic Web Conference ISWC-2020, 1. November - 6 November 2020, Virtual Conference
- **LIGER - Link Discovery with Partial Recall** by Kleanthi Georgala, Mohamed Ahmed Sherif and Axel-Cyrille Ngonga Ngomo in Proceedings of the 15th International Workshop on Ontology Matching 2020 (OM-2020), collocated with the 19th International Semantic Web Conference ISWC-2020, 1. November - 6 November 2020, Virtual Conference
- **LIMES - A Framework for Link Discovery on the Semantic Web** by Axel-Cyrille Ngonga Ngomo, Mohamed Ahmed Sherif, Kleanthi Georgala, Mofeed Hassan, Kevin Dreßler, Klaus Lyko, Daniel Obraczka, and Tommaso Soru. KI-Künstliche Intelligenz, German Journal of Artificial Intelligence - Organ des Fachbereichs "Künstliche Intelligenz" der Gesellschaft für Informatik e.V. (2021)
- **Systematic Survey on String Similarity Joins for Link Discovery** by Kleanthi Georgala and Axel-Cyrille Ngonga Ngomo (under review for the Journal of Web Semantics)
- **Using Machine Learning for Link Discovery on the Web of Data** by Axel-Cyrille Ngonga Ngomo, Daniel Obraczka, and Kleanthi Georgala. Demos at the European Conference on Artificial Intelligence. 2016.
- **Spam Filtering: an Active Learning Approach using Incremental Clustering** by Kleanthi Georgala, Aris Kosmopoulos, and George Paliouras. 2014. In Proceedings of

the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14) (WIMS '14). ACM, New York, NY, USA, Article 23, 12 pages.

- **Record linkage in medieval and early modern text** by Kleanthi Georgala, Benjamin van der Burgh , Marvin Meeng and Arno Knobbe. (2015). In Population Reconstruction (pp. 173-195). Springer, Cham.

.3 Curriculum Vitae

Kleanthi Georgala
Filonos 29B
172 36, Dafni, Greece
(+30) 6982786194
georgala@informatik.uni-leipzig.de
`http://aksw.org/KleanthiGeorgala.html`

Personal Data

Name: Kleanthi Georgala
Birth date: February 15th, 1989
Birth place: Cholargos, Greece
Nationality: Greek

Education & Work

2020 - Present
Intracom Telecom (Attica, Greece)
Product Marketing Engineer / Data Scientist

2018 - 2020
University of Paderborn (Paderborn, Germany)
Ph.D., Faculty for Computer Science, Electrical Engineering and Mathematics, Department of Computer Science.
Thesis title: *Fast and Scalable Link Discovery for Modern Data-Driven Applications.*

2015 - 2018
University of Leipzig (Leipzig, Germany)
Ph.D., Faculty of Mathematics and Computer Science, Department of Computer Science.
Thesis title: *Fast and Scalable Link Discovery for Modern Data-Driven Applications.*

2015 - 2014
Leiden University (Leiden, Netherlands)
Scientific Programmer, Leiden Institute of Advanced Computer Science, Faculty of Science

2014
Leiden University (Leiden, Netherlands)
Guest Lecturer at the Data Mining course (3rd year-Bachelor), Leiden Institute of Advanced Computer Science, Faculty of Science

2014
Leiden University (den Haag, Netherlands)
Instructor at "Big Data for Peace" Summer School for Peace Informatics, Leiden Institute of Advanced Computer Science, Faculty of Science

2014
University of Edinburgh (Edinburgh, United Kingdom)
Teaching Tutor at "Informatics 2B - Algorithms, Data Structures and Learning", College of Science and Engineering, School of Informatics

2012 - 2013
University of Edinburgh (Edinburgh, United Kingdom)
MSc in Artificial Intelligence, College of Science and Engineering, School of Informatics
MSc Thesis: Relevance Feedback with minimal training Data

2009-2011
Institute of Informatics and Telecommunications of the NCSR "Demokritos" (Athens, Greece)
BSc thesis: Active learning spam filter with incremental clustering.

2006-2011
National and Kapodistrian University of Athens (Athens, Greece)
BSc (Ptychion) in Informatics and Telecommunications, School of Science, Department of Informatics and Telecommunications
Grade: 7.97/10 - Very Good

2006
Athens Model School "Protipo" (Athens, Greece)
"Apolytyrion of Lyceum"
GPA : 19.2/20.0 - Excellent

Awards and Nominations

- Best Student Paper Award WI 2017, IEEE/WIC/ACM International Conference on Web Intelligence 2017
  for **An Evaluation of Models for Runtime Approximation in Link Discovery** by Kleanthi Georgala, Michael Hoffmann, and Axel-Cyrille Ngonga Ngomo
  Leipzig, Germany

- ISWC 2016 Travel Award, International Semantic Web Conference
  for **Scalable Link Discovery for Modern Data-Driven Applications** by Kleanthi Georgala
  Kobe, Japan

- Informatics UK/EU Master's Scholarship, University of Edinburgh
  Informatics UK/EU Master's Scholarship - University of Edinburgh for MSc Degree in Artificial Intelligence
  Edinburgh, United Kingdom

- High School - Lyceum Award
  Awards for being at the 10% top students of the school
  Athens, Greece

Research Interests

- Semantic Web
- Artificial Intelligent
- Machine Learning
- Data Mining

Technical and Programming Skills

- **Programming Languages Skills:**
  - C++, Java, Python
  - C,
  - HTML, PHP
  - Prolog
  - Matlab, R
  - SQL, SPARQL
  - Bash
- **Software Applications**
  - Netbeans, Eclipse, Weka
  - Oracle, MySQL, MongoDB, SQL Server
  - Kibana
  - Various triple stores: Virtuoso, Graph DB, Apache Jena Fuseki, Blazegraph
  - LateX, OpenOffice.org, Microsoft Office, LibreOffice
- **Operating Systems**
  - Ubuntu Linux
  - Unix
  - Microsoft Windows
  - Mac OS

Projects

- **HOBBIT**: `aksw.org/Projects/HOBBIT.html`
  Holistic Benchmarking of Big Linked Data.
  Funded by EU H2020 Research and Innovation Program
  Duration: 12/2015–11/2018
- **LIMES**: `http://aksw.org/Projects/LIMES`
  Link discovery framework for metric spaces.
- **SAKE**: `http://aksw.org/Projects/SAKE.html`
  With RDF and Machine Learning Getting Results Faster.
  Funded by BWMi (Federal Ministry for Economic Affairs and Energy)
  Duration: 12/2014 - 12/2017
- **Traces Through Time**: `https://gtr.ukri.org/projects?ref=AH%2FL010186%2F1`
  A new tool for finding linked records across our collections.
  Funded by AHRC (Arts and Humanities Research Council, United Kingdom)
  Duration: 01/2014 - 03/2015

Language Skills

- **Greek**: Native
- **English**: Advanced (Proficiency of Cambridge)
- **German**: Advanced (C1 Certificate)
- Familiar with **Dutch** and **Spanish**.

Research Community Service

- **Chair**: MOCHA2018 Challenge (ESWC 2018),
- **Program Committee**: GeoLD2018 (ESWC 2018), KESW 2017, MOCHA2017 Challenge (ESWC 2017), CSCUBS 2016, KESW 2016, NLIWoD 2015, AISI 2015
- **Organizer** for MOCHA2018 and MOCHA2017 Challenges at ESWC
- **Reviewer** for GeoLD2018 (ESWC 2018), CSCUBS 2016,NLIWoD 2015, AISI 2015
- **Presenter** for *ESWC 2018*, *WI 2017*, *ISWC 2016*, *ECAI 2016*, *WIMS 2014*
- Founding member and translator of FOSS UoA, the Free and Open Source Software community of University of Athens