HELIOS – Execution Optimization for Link Discovery

Axel-Cyrille Ngonga Ngomo[1]

Department of Computer Science
University of Leipzig
Johannisgasse 26, 04103 Leipzig
ngonga@informatik.uni-leipzig.de,
WWW home page: http://limes.sf.net

Abstract. Links between knowledge bases build the backbone of the Linked Data Web. In previous works, the combination of the results of time-efficient algorithms through set-theoretical operators has been shown to be very time-efficient for Link Discovery. However, the further optimization of such link specifications has not been paid much attention to. We address the issue of further optimizing the runtime of link specifications by presenting HELIOS, a runtime optimizer for Link Discovery. HELIOS comprises both a rewriter and an execution planner for link specifications. The rewriter is a sequence of fixed-point iterators for algebraic rules. The planner relies on time-efficient evaluation functions to generate execution plans for link specifications. We evaluate HELIOS on 17 specifications created by human experts and 2180 specifications generated automatically. Our evaluation shows that HELIOS is up to 300 times faster than a canonical planner. Moreover, HELIOS' improvements are statistically significant.

1 Introduction

Link Discovery (LD) plays a central role in the realization of the Linked Data paradigm. Several frameworks such as LIMES [9] and SILK [5] have been developed to address the time-efficient discovery of links. These frameworks take a *link specification* (short: LS, also called linkage rule [5]) as input. Each LS is converted internally into a sequence of operations which is then executed. While relying on time-efficient algorithms (e.g., PPJoin+ [17] and $\mathcal{HR}^3$ [7]) for single operations has been shown to be very time-efficient [9], the optimization of the execution of whole LS within this paradigm has been payed little attention to.

In this paper, we address this problem by presenting HELIOS, the (to the best of our knowledge) first execution optimizer for LD. HELIOS aims to reduce the costs necessary to execute a LS. To achieve this goal, our approach relies on two main components: a *rewriter* and a *planner*. The rewriter relies on algebraic operations to transform an input specification into an equivalent specification deemed less time-consuming to execute. The planner maps specifications to *execution plans*, which are sequences of operations from which a mapping results. HELIOS' planner relies on time-efficient evaluation func-

tions to generate possible plans, approximate their runtime and return the one that is likely to be most time-efficient.[1] Our contributions are:

1. We present a novel generic representation of LS as bi-partite trees.
2. We introduce a novel approach to rewriting LS efficiently.
3. We explicate a novel planning algorithm for LS.
4. We evaluate HELIOS on 2097 LS (17 manually and 2080 automatically generated) and show that it outperforms the state of the art by up to two orders of magnitude.

The rest of this paper is structured as follows: First, we present a formal specification of LS and execution plans for LS. Then, we present HELIOS and its two main components. Then, we evaluate HELIOS against the state of the art. Finally, we give a brief overview of related work and conclude.

2 Formal Specification

In the following, we present a graph grammar for LS. We employ this grammar to define a *normal form* (NF) for LS that will build the basis for the rewriter and planner of HELIOS. Thereafter, we present execution plans for LS, which formalize the sequence of operations carried out by execution engines to generate links out of specifications. As example, we use the RDF graphs shown in Table 1, for which the perfect LD results is {(`ex1:P1`, `ex2:P1`), (`ex1:P2`, `ex2:P2`), (`ex1:P3`, `ex2:P3`), (`ex1:P4`, `ex2:P4`)}.

2.1 Normal Form for Link Specifications

Formally, most LD tools aim to discover the set $\{(s,t) \in S \times T : R(s,t)\}$ provided an input relation R (e.g., `owl:sameAs`), a set S of source resources (for example descriptions of persons) and a set T of target resources. To achieve this goal, declarative LD frameworks rely on LS, which describe the conditions under which $R(s,t)$ can be assumed to hold for a pair $(s,t) \in S \times T$. Several grammars have been used for describing LS in previous works [7, 5, 10]. In general, these grammars assume that LS consist of two types of atomic components: *similarity measures* m, which allow comparing property values of input resources and *operators* op, which can be used to combine these similarities to more complex specifications.

Without loss of generality, a similarity measure m can be defined as a function $m : S \times T \to [0,1]$. We use *mappings* $M \subseteq S \times T \times [0,1]$ to store the results of the application of a similarity function to $S \times T$ or subsets thereof. We also store the results of whole link specifications in mappings. The set of all mappings is denoted by $\mathcal{M}$. We call a measure *atomic* iff it relies on exactly one similarity measure σ (e.g., the edit similarity, dubbed `edit`)[2] to compute the similarity of a pair $(s,t) \in S \times T$

[1] HELIOS was implemented in the LIMES framework. All information to the tool can be found at http://limes.sf.net. A graphical user interface for the tool can be accessed via the SAIM interface at http://aksw.org/projects/SAIM.

[2] We define the edit similarity of two strings s and t as $(1 + lev(s,t))^{-1}$, where lev stands for the Levenshtein distance.

Table 1. Examplary graphs.

Persons1 graph	Persons2 graph
`ex1:P1 ex:label "Anna"@en .`	`ex2:P1 ex:label "Ana"@en .`
`ex1:P1 ex:age "12"^^xsd:integer .`	`ex2:P1 ex:age "12"^^xsd:integer .`
`ex1:P1 a ex:Person .`	`ex2:P1 a ex:Person .`
`ex1:P2 ex:label "Jack"@en .`	`ex2:P2 ex:label "Jack"@en .`
`ex1:P2 ex:age "15"^^xsd:integer .`	`ex2:P2 ex:age "14"^^xsd:integer .`
`ex1:P2 a ex:Person .`	`ex2:P2 a ex:Person .`
`ex1:P3 ex:label "John"@en .`	`ex2:P3 ex:label "Joe"@en .`
`ex1:P3 ex:age "16"^^xsd:integer .`	`ex2:P3 ex:age "16"^^xsd:integer .`
`ex1:P3 a ex:Person .`	`ex2:P3 a ex:Person .`
`ex1:P4 ex:label "John"@en .`	`ex2:P4 ex:label "John"@en .`
`ex1:P4 ex:age "19"^^xsd:integer .`	`ex2:P4 ex:age "19"^^xsd:integer .`
`ex1:P4 a ex:Person .`	`ex2:P4 a ex:Person .`

with respect to the (list of) properties p_s of s and p_t of t and write $m = \sigma(p_s, p_t)$. A similarity measure m is either an atomic similarity measure or the combination of two similarity measures via a *metric operator* such as $\max$, $\min$ or linear combinations. For example, `edit`(s.label, t.label) is an atomic measure while $\max$(`edit`(s.label, t.label), `edit`(s.age, t.age)) is a complex similarity measure.

We define a *filter* as any function which maps a mapping M to another mapping M'. *Similarity filters* $f(m, \theta)$ return $f(m, \theta, M) = \{(s, t, r')|\exists r : (s, t, r) \in M \wedge m(s, t) \geq \theta \wedge r' = \min\{m(s, t), r\}\}$. *Threshold filters* $i(\theta)$ return $i(\theta, M) = \{(s, t, r) \in M : r \geq \theta\}$. Note that $i(0, M) = M$ and that we sometimes omit M from similarity filters for the sake of legibility.

We call a specification *atomic* when it consists of exactly one filtering function. For example, applying the atomic specification f(edit(`ex:label`, `ex:label`), 1) to our input data leads to the mapping {(`ex1:P3`, `ex2:P4`, 1), (`ex1:P2`, `ex2:P2`, 1), (`ex1:P4`, `ex2:P4`, 1)}. A complex specification can be obtained by combining two specifications L_1 and L_2 by (1) a *mapping operator* (that allows merging the mappings which result from L_1 and L_2) and (2) a subsequent filter that allows postprocessing the results of the merging.[3] In the following, we limit ourselves to the operators based on $\cup$, $\cap$ and $\setminus$ (set difference), as they are sufficient to describe any operator based on set operators. We extend these operators to mappings as follows:

- $M_1 \cap M_2 = \{(s, t, r) : \exists a, b\ (s, t, a) \in M_1 \wedge (s, t, b) \in M_2 \wedge r = \min(a, b)\}$.
- $M_1 \cup M_2 = \{(s, t, r) : (\neg\exists(s, t, a) \in M_1 \wedge (s, t, r) \in M_2) \vee (\neg\exists(s, t, b) \in M_2 \wedge (s, t, r) \in M_1) \vee (\exists(s, t, a) \in M_1 \wedge \exists(s, t, b) \in M_2 \wedge r = \max(a, b))\}$.
- $M_1 \setminus M_2 = \{(s, t, r) \in M_1 : \neg\exists(s, t, a) \in M_2\}$.

For example, if M_1 = {(`ex1:P1`, `ex2:P2`, 1), (`ex1:P1`, `ex2:P3`, 1)} and M_2 = {(`ex1:P1`, `ex2:P2`, 0.5)} then $M_1 \cup M_2 = M_1$, $M_1 \cap M_2 = M_2$ and $M_1 \setminus M_2$ = {(`ex1:P1`, `ex2:P3`, 1)}.

Based on this grammar, we can regard all LS as *bi-partite directed trees* $L = (V(L), E(L))$ which abide by the following restrictions:

[3] We rely on binary operators throughout this paper because n-ary set operators can always be mapped to a sequence of binary operators.

1. The vertices of L can be either *filter nodes* $f \in F$ or *operator nodes* $op \in OP$, i.e., $V(L) = F \cup OP$. The leaves and the root of L are always filter nodes. The leaves are filters that run on $S \times T$.
2. Edges in L can only exist between filters and operators, i.e., $E(L) \subseteq (F \times OP) \cup (OP \times F)$.

An example of a LS is shown in Figure 1. We call this representation of LS their NF. In the rest of this paper, we deal exclusively with finite specifications, i.e., specifications such that their NF contains a finite number of nodes. We call the number of filter nodes of a specification L the *size* of L and denote it $|L|$. For example, the size of the specification in Figure 1 is 3. We dub the direct child of L's root the *operator of* L. For example, the operator of the specification in Figure 1 is $\cap$. We call a LS L' a *sub-specification* of L (denoted $L' \subseteq L$) if L''s NF is a sub-tree of L's NF that abides by the definition of a specification (i.e., if the root of L''s NF is a filter node and the NF of L' contains all children of L' in L). For example, $f(\texttt{edit}(label, label), 0.3)$ is a sub-specification of our example. We call a L' a *direct sub-specification* of L (denoted $L' \subset_1 L$) if L' is a sub-specification of L whose root node a grandchild of the L's root. For example, $f(\texttt{edit}(label, label), 0.3)$ is a direct sub-specification of the LS shown in Figure 1. Finally, we transliterate LS by writing $f(m, \theta, op(L_1, L_2))$ where $f(m, \theta)$ is L's root, *op* is L's operator, $L_1 \subset_1 L$ and $L_2 \subset_1 L$.

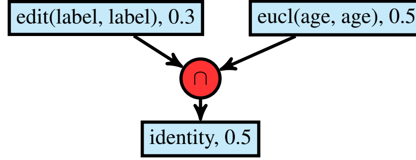


Fig. 1. A LS for linking the datasets Person1 and Person2. The filter nodes are rectangles while the operator nodes are circles. eucl(s.age, t.age) = $(1 + |s.age - t.age|)^{-1}$. This LS can be transliterated $i(\cap(f(\texttt{edit}(label, label), 0.3), f(\texttt{eucl}(age, age), 0.5)), 0.5)$.

2.2 Execution Plans

We define an execution plan P as a sequence of *processing steps* $p_1, ..., p_n$ of which each is drawn from the set $\mathcal{A} \times \aleph \times \mathcal{T} \times \mathcal{M} \times \mathcal{M}$, where:

1. $\mathcal{A}$ is the set of all *actions* that can be carried out. This set models all the processing operations that can be carried out when executing a plan. These are:
   (a) run, which runs the computation of filters $f(m, \theta)$ where m is an atomic measure. This action can make use of time-efficient algorithms such as $\mathcal{HR}^3$.
   (b) filter, which runs filters $f(m, \theta)$ where m is a complex measure.
   (c) filterout, which runs the negation of $f(m, \theta)$.
   (d) Mapping operations such as union, intersection and minus (mapping difference) and

(e) `return`, which terminates the execution and returns the final mapping.
The result of each action (and therewith of each processing step) is a mapping.
2. $\aleph$ is the set of all complex measures as described above united with the $\emptyset$-measure, which is used by actions that do not require measures (e.g., `return`).
3. $\mathcal{T}$ is the set of all possible thresholds (generally $[0, 1]$) united with the $\emptyset$-threshold for actions that do not require any threshold (e.g., `union`) and
4. $\mathcal{M}$ is the set of all possible mappings, i.e., the powerset of $S \times T \times [0, 1]$.

We call the plan P atomic if it consists of exactly one processing step. An *execution planner* EP is a function which maps a LS to an execution plan P. The *canonical planner* EP_0 is the planner that runs specification in *postorder*, i.e., by traversing the NF of LS in the order left-right-root. The approach currently implemented by LIMES [9] is equivalent to EP_0. For example, the plan generated by EP_0 for Figure 1 is shown in the left column of Table 2. For the sake of brevity and better legibility, we will use abbreviated versions of plans that do not contain $\emptyset$ symbols. The abbreviated version of the plan generated by EP_0 for the specification in Figure 1 is shown in the right column of Table 2. We call two plans *equivalent* when they return the same results for all possible S and T. We call a planner *complete* when it always returns plans that are equivalent to those generated by EP_0.

Table 2. Plans for the specification shown in Figure 1

Canonical Plan	**Abbreviated Canonical Plan**
M_1=(run,edit(label,label),0.3,$\emptyset$,$\emptyset$) M_2=(run,eucl(age,age),0.5,$\emptyset$,$\emptyset$) M_3=(intersection,$\emptyset$,$\emptyset$,M_1,M_2) M_4=(return,$\emptyset$,$\emptyset$,M_3,$\emptyset$)	M_1=(run,edit(label,label),0.3) M_2=(run,eucl(age,age),0.5) M_3=(intersection,M_1,M_2) M_4=(return,M_3)
Alternative Plan1 (abbreviated)	**Alternative Plan2** (abbreviated)
M_1=(run,edit(label,label),0.3) M_2=(filter,eucl(age,age),0.5,M_1) M_3=(return,M_2)	M_1=(run,eucl(age,age),0.5) M_2=(filter,edit(label,label),0.3,M_1) M_3=(return,M_2)

The insight behind our paper is that equivalent plans can differ significantly with respect to their runtime. For example, the canonical plan shown in Table 2 would lead to 32 similarity computations (16 for `edit` and 16 for `euclidean`) and one mapping intersection, which can be computed by using 16 lookups. If we assume that each operation requires 1ms, the total runtime of this plan would be 48ms. The alternative plan 1 shown in Table 2 is equivalent to the plans in Table 2 but only runs 16 computations of `edit` (leading to M_1 of size 6) and 6 computations of `euclidean` on the data contained in M_1. The total runtime of this plan would thus be 22ms. Detecting such *runtime-efficient* and *complete plans* is the goal of HELIOS.

3 HELIOS

HELIOS is an optimizer for LS which consists of two main components: a rewriter (denoted RW) and a planner (denoted HP). Each LS L to be processed is first forwarded

to RW, which applies several algebraic transformation rules to transform L into an equivalent LS L' that promises to be more efficient to execute. The aim of HP is then to derive a *complete plan* P for L'. This plan is finally sent to the execution engine, which runs the plan and returns a final mapping. In the following, we present each of these components.[4] Throughout the formalization, we use $\rightarrow$ for logical implications and $\Rightarrow$ to denote rules.

3.1 The HELIOS Rewriter

RW implements an iterative rule-based approach to rewriting. Each iteration consists of three main steps that are carried out from leaves towards the root of the input specification. In the first step, sub-graphs of the input specification L are replaced with equivalent sub-graphs which are likely to be more efficient to run. In a second step, dependency between nodes in L are determined and propagated. The third step consists of removing portions of L which do not affect the final results of L's execution. These three steps are iterated until a fixpoint is reached.

Step 1: Rewriting Given a LS L, RW begins by rewriting the specification using algebraic rules dubbed *leaf generation rules*.

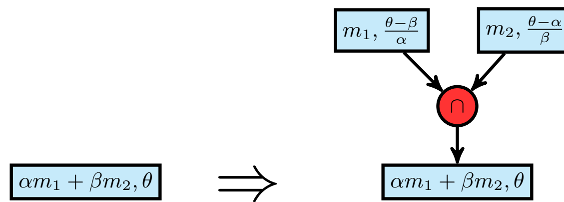


Fig. 2. Leaf generation rule for linear combinations.

The leaf generation rules ($\mathcal{LR}$) make use of relations between metric operators and specification operators to transform leaf nodes with complex measures into graphs whose leaves contain exclusively atomic measures. For example, the rule shown in Figure 2 transforms a filter that relies on the linear combinations of 2 measures into a LS with three filters whose leaves only contain atomic measures as described in [9]. While it might seem absurd to alter the original filter in this manner, the idea here is that we can now run specialized algorithms for m_1 and m_2, then compute the intersection M of the resulting mapping and finally simply check each of the (s,t) with $\exists r : (s,t,r) \in M$ for whether it abides by the linear combination in the root filter. This approach is usually more time-efficient than checking each $(s,t) \in S \times T$ for whether it abides by the linear combination in the original specification. Similar rules can be devised for $\min$ (see Figure 3), $\max$ and the different average functions used in LD frameworks. After L has been rewritten by the rules in $\mathcal{LR}$, each of its leaves is a filter with atomic measures.

[4] Due to space restrictions, some of the details and proofs pertaining to the rewriter and planner are omitted. Please consult http://limes.sf.net for more details.

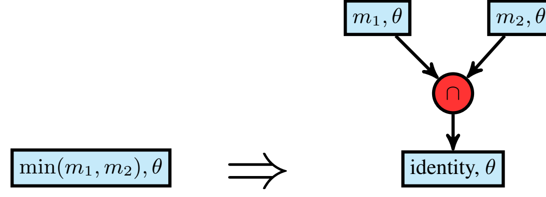


Fig. 3. Rule for minimum. In the corresponding rule for maximum, the mapping union is used.

Step 2: Dependency Detection and Propagation The idea behind the use of *dependencies* is to detect and eliminate redundant portions of the specification. Consequently, RW implements two types of dependency-based rules: *dependency detection rules* and *dependency propagation rules*. Formally, we say that L_1 *depends* on L_2 (denoted $depends(L_1, L_2)$) if the mapping resulting from L_1 is a subset of the mapping resulting from L_2 for all possible S and T. RW generates dependencies between leaves (which now only contain atomic measures) by making use of

$$L_1 = f(m, \theta_1) \wedge L_2 = f(m, \theta_2) \wedge \theta_1 \geq \theta_2 \Rightarrow depends(L_1, L_2). \tag{1}$$

Moreover, RW makes use of dependencies have been shown to apply between several similarity and distance measures that are commonly used in literature. For example,the authors of [17] show that for two non-empty strings x and y, $\texttt{jaccard}(x, y) \geq \theta \rightarrow \texttt{overlap}(x, y) \geq \frac{\theta}{1+\theta}(|x| + |y|)$. Given that $|x| \geq 1$ and $|y| \geq 1$, we can infer that

$$\texttt{jaccard}(x, y) \geq \theta \rightarrow \texttt{overlap}(x, y) \geq \frac{2\theta}{1+\theta}. \tag{2}$$

Thus, if $L_1 = f(\texttt{jaccard}(p_s, p_t), \theta_1)$ and $L_2 = f(\texttt{overlap}(p_s, p_t), \theta_2)$ with $\theta_2 \leq \frac{2\theta_1}{1+\theta_1}$, then $depends(L_1, L_2)$ holds. Currently, RW implements dependencies between the `overlap`, `trigrams` and the `jaccard` similarities discussed in [17].

Leaf-level dependencies can be propagated towards the root of the specification based on the following rules:

- p_1: $L = i(\theta, op(L_1, L_2)) \wedge L_1 = f(m, \theta_1, op_1(L_{11}, L_{12})) \wedge L_2 = f(m, \theta_2, op_2(L_{21}, L_{22})) \wedge \theta_1 \geq \theta \wedge \theta_2 \geq \theta) \Rightarrow L := i(0, op(L_1, L_2))$ (if the threshold of the father of any operator is smaller than that of all its children and the father node is an identity filter, then the threshold of the father can be set to 0).
- p_2: $depends(L_1, L') \wedge depends(L_2, L') \wedge L = f(m, \theta, \cap(L_1, L_2)) \Rightarrow depends(L, L')$ (if all children of a conjunction depend on L' then the father of this conjunction depends on L').
- p_3: $L = f(m, 0, \cup(L_1, L_2)) \wedge (depends(L', L_1) \vee depends(L', L_2)) \Rightarrow depends(L', L)$ (if L' depends on one child of a disjunction and the father of the disjunction has the threshold 0 then L' depends on the father of the disjunction).

Step 3: Reduction Given two specifications $L_1 \subset_1 L$ and $L_2 \subset_1 L$ with $depends(L_1, L_2)$, we can now *reduce* the size of $L = filter(m, \theta, op(L_1, L_2))$ by using the following rules:

r_1: $L' = filter(m, \theta, \cap(L_1, L_2)) \wedge depends(L_1, L_2) \Rightarrow L' := filter(m, \theta, L_1))$,
r_2: $L' = filter(m, \theta, \cup(L_1, L_2)) \wedge depends(L_1, L_2) \Rightarrow L' := filter(m, \theta, L_2))$,
r_3: $L' = filter(m, \theta, \backslash(L_1, L_2)) \wedge depends(L_1, L_2) \Rightarrow L' := \emptyset$ where := stands for overwriting.

An example that elucidates the ideas behind $\mathcal{DR}$ is given in Figure 4. Set operators applied to one mapping are assumed to not alter the mapping.

The leaf generation terminates after at most as many iterations as the total number of atomic specifications used across all leaves of the input LS L. Consequently, this step has a complexity of $O(|L'|)$ where $L' = \mathcal{LR}(L)$. The generation of dependencies requires $O(|L'|^2)$ node comparisons. Each time a reduction rule is applied, the size of the L' decreases, leading to reduction rules being applicable at most $|L'|$ times. The complexity of the reduction is thus also $O(|L'|)$. In the worst case of a left- or right-linear specification, the propagation of dependencies can reach the complexity $O(|L'|^2)$. All three steps of each iteration thus have a complexity of at most $O(|L|'^2)$ and the specification is at least one node smaller after each iteration. Consequently, the worst-case complexity of the rewriter is $O(|L'|^3)$.

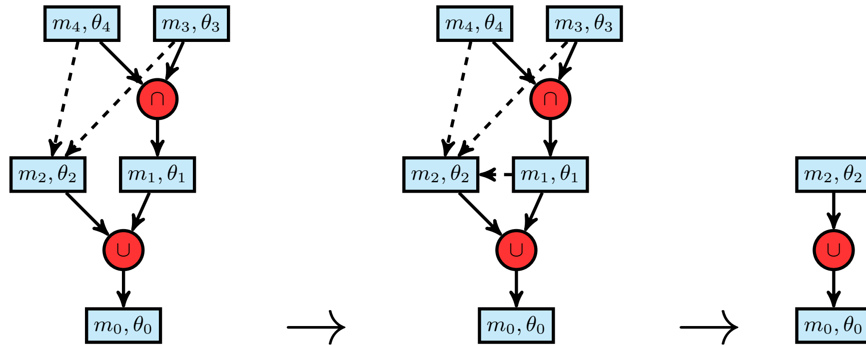


Fig. 4. Example of propagation of dependencies. The dashed arrows represent dependencies. The dependencies from the left figure are first (using rule p_1). Then, the reduction rule r_2 is carried out, leading to the specification on the right.

3.2 The HELIOS Planner

The goal of the HELIOS planner HP is to convert a given LS into a plan. Previous work on query optimization for databases have shown that finding the optimal plan for a given query is exponential in complexity [15]. The complexity of finding the perfect plan for a LS is clearly similar to that of finding a play for a given query. To circumvent the complexity problem, we rely on the following optimality assumption: Given $L_1 \subset_1 L$ and $L_2 \subset_1 L$ with $L = f(m, \theta, op(L_1, L_2))$, a good plan for L can be derived from plans for L_1 and L_2. In the following, we begin by explaining core values that HP needs to evaluate a plan. In particular, we explain how HP evaluates atomic and complex plans. Thereafter, we present the algorithm behind HP and analyze its complexity.

Plan Evaluation HP uses two values to characterize any plan P: (1) the approximate runtime of P (denoted $\gamma(P)$) and (2) the selectivity of P (dubbed $s(P)$), which encodes the size of the mapping returned by P as percentage of $|S \times T|$.

Computing $\gamma(P)$ and $s(P)$ for atomic LS: Several approaches can be envisaged to achieve this goal. In our implementation of HP, we used approximations based on sampling. The basic assumption behind our feature choice was that LD frameworks are usually agnostic of S and T before the beginning of the LD. Thus, we opted for approximating the runtime of atomic plans P by using $|S|$ and $|T|$ as parameters. We chose these values because they be computed in linear time.[5] To approximate $\gamma(P)$ for atomic plans, we generated source and target datasets of sizes $1000, 2000, \ldots, 10000$ by sampling data from the English labels of DBpedia 3.8. We then stored the runtime of the measures implemented by our framework for different thresholds θ between 0.5 and 1.[6] The runtime of the ith experiment was stored in the row y_i of a column vector Y. The corresponding experimental parameters $(1, |S|, |T|, \theta)$ were stored in the row r_i of a four-column matrix R. Note that the first entry of all r_i is 1 to ensure that we can learn possible constant factors. We finally computed the vector $\Gamma = (\gamma_0, \gamma_1, \gamma_2, \gamma_3)^T$ such that

$$\gamma(P) = \gamma_0 + \gamma_1|S| + \gamma_2|T| + \gamma_3\theta. \tag{3}$$

To achieve this goal, we used the following exact solution to linear regression: $\Gamma = (R^T R)^{-1} R^T Y$. The computation of $s(P)$ was carried out similarly with the sole difference that the entries y_i for the computation of $s(P)$ were $\frac{|M_i|}{|S| \times |T|}$, where M_i is the size of the mapping returned by the ith experiment. Figure 5 shows a sample of the results achieved by different algorithms in our experiments. The plan returned for the atomic $LSf(m, \theta)$ is `(run,m,`θ`)`.

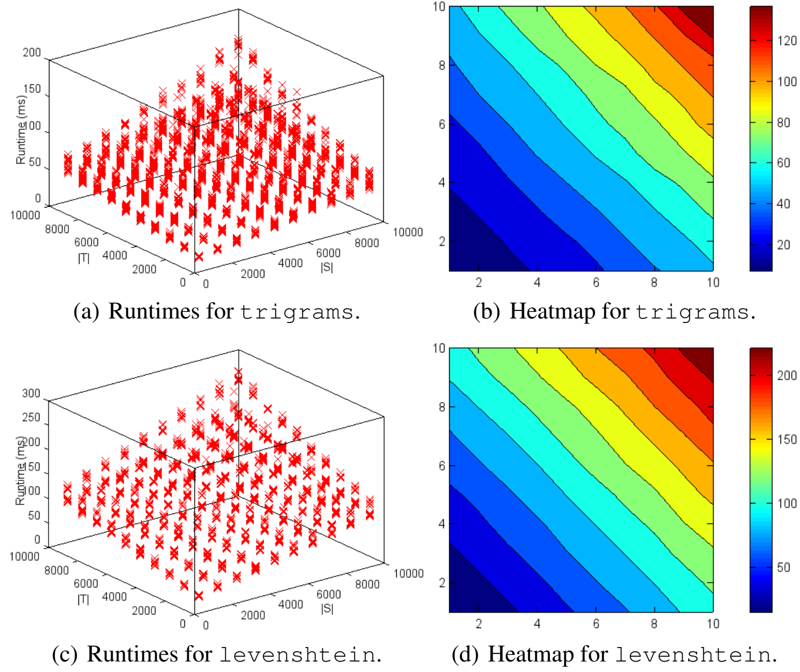
Computing $\gamma(P)$ and $s(P)$ for complex LS: The computation of the costs associated with atomic `filter`, `filterout` and operators was computed analogously to the computation of runtimes for atomic LS. For filters, the feature was the size of the input mapping. For non-atomic plans P, we computed $\gamma(P)$ by summing up the $\gamma(p_i)$ for all the steps p_i included in the plan. The selectivity of operators was computed based on the selectivity of the mappings that served as input for the operators. To achieve this goal, we assumed that the selectivity of a plan P to be the probability that a pair (s, t) is returned after the execution of P. Moreover, we assumed the input mappings M_1 (selectivity: s_1) resp. M_2 (selectivity: s_2) to be the results of independent computations. Based on these assumptions, we derived the following selectivities for $op(M_1, M_2)$:

- $op = \cap \rightarrow s(op) = s_1 s_2$.
- $op = \cup \rightarrow s(op) = 1 - (1 - s_1)(1 - s_2)$.
- $op = \setminus \rightarrow s(op) = s_1(1 - s_2)$.

---

[5] Other values can be used for this purpose but our results suggest that using $|S|$ and $|T|$ is sufficient in most cases.

[6] We used the same hardware as during the evaluation.



(a) Runtimes for `trigrams`. (b) Heatmap for `trigrams`.

(c) Runtimes for `levenshtein`. (d) Heatmap for `levenshtein`.

Fig. 5. Runtimes achieved by PPJoin+ (`trigrams`) and EDJoin (`levenshtein`) for $\theta = 0.5$. The x-axis of the heatmap show $|S|$ in thousands, while the y-axis shows $|T|$ in thousands. The color bars show the runtime in ms.

The HP algorithm The core of the approach implemented by HP is shown in Algorithm 1. For atomic specifications $f(m, \theta)$, HP simply returns `(run,m,`θ`)` (GETBESTPLAN method in Algorithm 1). If different algorithms which allow running m efficiently are available, HP chooses the implementation that leads to the smallest runtime $\gamma(P)$. Note that the selectivity of all algorithms that allow running m is exactly the same given that they must return the same mapping. If the specification $L = (m, \theta, op(L_1, L_2))$ is not atomic, HP's core approach is akin to a divide-and-conquer approach. It first devises a plan for L_1 and L_2 and then computes the costs of different possible plans for *op*. For $\cap$ for example, the following three plans are equivalent:

1. **Canonical plan.** This plan simply consists of merging (via the CONCATENATE method in Algorithm 1) the results of the best plans for L_1 and L_2. Consequently, the plan consists of (1) running the best plan for L_1 (i.e., Q_1 in Algorithm 1), (2) running the best plan for L_2(i.e., Q_2 in Algorithm 1), then (3) running the `intersection` action over the results of Q_1 and Q_2 and finally (4) running `filter` over the result of the intersection action.
2. **Filter-right plan.** This plan uses $f(m_2, \theta_2)$ as a filter over the results of Q_1. Consequently, the plan consists of (1) running the best plan for L_1, then (2) running the `filter` action with measure m_2 and threshold θ_2 over the results of Q_1 and finally (3) running `filter` with measure m and threshold θ over the previous result.

3. **Filter-left plan.** Analogous to the filter-right plan with L_1 and L_2 reversed.

Similar approaches can be derived for the operators $\cup$ and $\setminus$ as shown in Algorithm 1. HP now returns the least costly plan as result (GETLEASTCOSTLY method in Algorithm 1). This plan is finally forwarded to the execution engine which runs the plan and returns the resulting mapping.

Given that the alternative plans generated by HP are equivalent and that HP always chooses one of this plan, our algorithm is guaranteed to be complete. Moreover, HP approximates the runtime of at most 3 different plans per operator and at most k different plans for each leaf of the input specification (where k is the maximal number of algorithms that implements a measure m in our framework). Consequently, the runtime complexity of HP is $O(\max\{k, 3\} \times |L|)$.

Algorithm 1 The PLAN method

```
if L is atomic then
    P = getBestPlan(L);
else
    if L = f(m, θ, op(L1)) then
        P := getBestPlan(L1)
    else
        Q1 := plan(L1)
        Q2 := plan(L2)
        if L = f(m, θ, ∩(L1, L2)) then
            P0 := concatenate(intersection, Q1, Q2)
            P1 := concatenate(filter(m1, θ1), Q2)
            P2 := concatenate(filter(m2, θ2), Q1)
            P := getLeastCostly(P0, P1, P2)
        else if L = f(m, θ, ∪(L1, L2)) then
            P0 := concatenate(union, Q1, Q2)
            P1 := concatenate(union, filter(m2, θ2, S × T), Q2)
            P2 := concatenate(union, filter(m1, θ1, S × T), Q1)
            P := getLeastCostly(P0, P1, P2)
        else if L = f(m, θ, \(L1, L2)) then
            P0 := concatenate(minus, Q1, Q2)
            P1 := concatenate(filterout(m2, θ2), Q2)
            P := getLeastCostly(P0, P1)
        end if
    end if
    a0 = filter(m, θ)
    P = concatenate(a0, P)
end if
return P
```

4 Evaluation

4.1 Experimental Setup

The aim of our evaluation was to measure the runtime improvement of HELIOS the overall runtime of LS. We thus compared the runtimes of EP_0 (i.e., LIMES), RW (i.e., RW + EP_0), HP and HELIOS (i.e., RW +HP) in our experiments. We chose LIMES because it has been shown to be very time-efficient in previous work [9]. We considered manually created and automatically generated LS. All experiments were carried out on

server running Ubuntu 12.04. In each experiment, we used a single kernel of a 2.0GHz AMD Opteron processor with 10GB RAM.

The manually created LS were selected from the LATC repository.[7] We selected 17 LS which relied on SPARQL endpoints that were alive or on data dumps that were available during the course of the experiments. The specifications linked 18 different datasets and had sizes between 1 and 3. The small sizes were due to humans tending to generate small and non-redundant specifications.

The automatic specifications were generated during a single run of specification learning algorithm EAGLE [8] on four different benchmark datasets described in Table 4.[8] The mutation and crossover rates were set to 0.6 while the number of inquiries per iteration was set to 10. The population size was set to 10. The sizes of the specifications generated by EAGLE varied between 1 and 11. We compared 1000 LS on the OAEI 2010 Restaurant and the DBLP-ACM dataset each, 80 specifications on the DBLP-Scholar dataset and 100 specifications on LGD-LGD. We chose to use benchmark datasets to ensure that the specifications used in the experiments were of high-quality w.r.t. the F-measure they led to. Each specification was executed 10 times. No caching was allowed. We report the smallest runtimes over all runs for all configurations to account for possible hardware and I/O influences.[9]

4.2 Results on Manual Specifications

The results of our experiments on manual specifications are shown in Table 3 and allow deriving two main insights: First, HELIOS can improve the runtime of atomic specifications (which made up 62.5% of the manual LS). This result is of tremendous importance as it suggests that the overhead generated by HELIOS is mostly insignificant, even for specifications which lead to small runtimes (e.g., DBP-DataGov requires 8ms). Moreover, our experiments reveal that HELIOS achieves a significant improvement of the overall runtime of specifications with sizes larger than 1 (37.5% of the manual LS). In the best case, HELIOS is 49.5 times faster than EP_0 and can reduce the runtime of the LS LDG-DBP (A) from 52.7s to 1.1s by using a filter-left plan. Here, we see that the gain in runtime generated by HELIOS grows with $|S| \times |T|$. This was to be expected as a good plan has more effect when large datasets are to be processed. Overall, HELIOS outperforms LIMES' canonical planner on all non-atomic specifications. On average, HELIOS is 4.3 times faster than the canonical planner on LS of size 3.

4.3 Results on Automatic Specifications

Overall, our results on automatic specifications show clearly that HELIOS outperforms the state of the art significantly. In Table 4, we show the average runtime of EP_0, RW, HP and HELIOS on four different datasets of growing sizes. The overall runtime of

[7] https://github.com/LATC/24-7-platform/tree/master/link-specifications

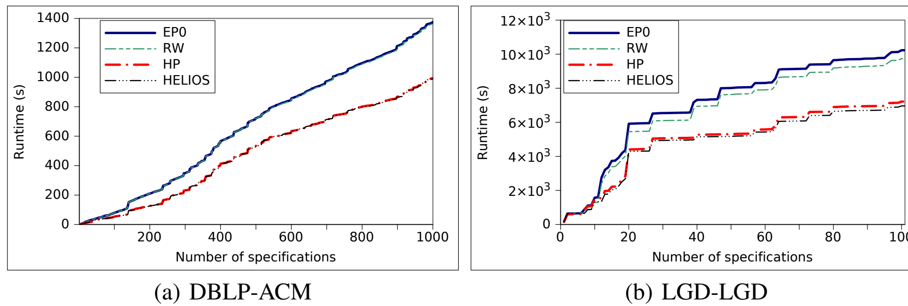
[8] The Restaurant data is available at http://oaei.ontologymatching.org/2010/. DBLP-ACM and DBLP-Scholar are at http://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution.

[9] All evaluation results can be found at https://github.com/AKSW/LIMES.

Table 3. Comparison of runtimes on manual specifications. The top portion of the table shows runtimes of specifications of size 1 while the bottom part shows runtimes on specifications of size 3. EVT stands for Eventseer, DF for DogFood, (P) stands for person, (A) stands for airports, (U) stands for universities, (E) stands for events. The best runtimes are in bold.

Source - Target	$\|S\| \times \|T\|$	EP_0 (ms)	RW (ms)	HP (ms)	HELIOS (ms)	Gain (ms)
DBP - Datagov	1.7×10^3	**8**	**8**	**8**	**8**	0
RKB - DBP	2.2×10^3	**1**	**1**	**1**	**1**	0
Epo - DBP	73.0×10^3	54	**53**	54	**53**	1
Rail - DBP	133.2×10^3	269	**268**	**268**	**268**	1
Stad - Rmon	341.9×10^3	25	23	15	**14**	11
EVT - DF (E)	531.0×10^3	**893**	906	909	905	-12
Climb - Rail	1.9×10^6	41	**40**	**40**	**40**	1
DBLP - DataSW	92.2×10^6	59	59	58	**54**	5
EVT - DF (P)	148.4×10^6	2,477	2,482	2,503	**2,434**	43
EVT - DBLP	161.0×10^6	9,654	**9,575**	9,613	9,612	42
DBP - OpenEI	10.9×10^3	**2**	**2**	**2**	**2**	0
DBP - GSpecies	94.2×10^3	120	**119**	120	**119**	1
Climb - DBP	312.4×10^3	**55**	**55**	**55**	**55**	0
DBP - LGD (E)	34.1×10^6	2,259	2,133	**1,206**	1,209	1,050
Climb - LGD	215.0×10^6	24,249	24,835	**3,497**	3,521	20,728
DBP - LGD (A)	383.8×10^6	52,663	59,635	1,066	**1,064**	51,599
LGD - LGD	509.3×10^9	46,604	38,560	32,831	**22,497**	24,107

HELIOS is clearly superior to that of EP_0 on all datasets. As expected, the gain obtained by using HELIOS grows with the size of $|S| \times |T|$. In particular, the results on the very small Restaurant dataset support the results achieved on the manual specifications. While HP alone does not lead to a significant improvement, HELIOS leads to an improvement of the overall runtime by 6.35%. This improvement is mostly due to RW eliminating filters and therewith altering the plans generated by HP. These alterations allow for shorter and more time-efficient plans.



(a) DBLP-ACM (b) LGD-LGD

Fig. 6. Cumulative runtimes on DBLP-ACM and LGD-LGD.

On the larger DBLP-ACM dataset, HELIOS achieve a runtime that is up to 185.8 times smaller than that of EP_0 (e.g., for $f(\cap(f($`jaccard`(authors, authors), 0.93),

Table 4. Summary of the results on on automatically generated specifications. $|L|$ shows for the average size $\pm$ standard deviation of the specifications in the experiment. F_1 shows the F-measure achieved by EAGLE on the dataset. The runtimes in four rightmost columns are the average runtimes in seconds.

	$\|S\| \times \|T\|$	$\|L\|$	F_1	EP_0	RW	HP	HELIOS
Restaurants	72.3×10^3	4.44±1.79	0.89	0.15	0.15	0.15	**0.14**
DBLP-ACM	6.0×10^6	6.61±1.32	0.99	1.38	1.37	1.00	**0.99**
DBLP-Scholar	168.1×10^6	6.42±1.47	0.91	17.44	17.41	13.54	**13.46**
LGD-LGD	5.8×10^9	3.54±2.15	0.98	102.33	97.40	72.19	**69.64**

f(`edit`(venue, venue), 0.93)),0.53)). Yet, given that the runtime approximations are generic, HELIOS sometimes generated plans that led to poorer runtimes. In the worst case, a plan generated by HELIOS was 6.5 times slower than the plan generated by EP_0 (e.g., for $f(\cap(f($`edit`(authors, authors), 0.59), $f($`cosine`(venue,venue),0.73)),0.4)). On average, HELIOS is 38.82% faster than EP_0. Similar results can be derived from DBLP-Scholar, where HELIOS is 29.61% faster despite having run on only 80 specifications. On our largest dataset, the time gain is even larger with HELIOS being 46.94% faster. Note that this improvement is very relevant for end users, as it translates to approximately 1h of runtime gain for each iteration of our experiments. Here, the best plan generated by HELIOS is 314.02 times faster than EP_0. Moreover, we can clearly see the effect of RW with average runtime improvement of 5.1% (see Figure 6).

We regard our overall results as very satisfactory given that the algorithms underlying EP_0 are in and of themselves already optimized towards runtime. Still, by combining them carefully, HELIOS can still cut down the overall runtime of learning algorithms and even of manually created link specifications. To ensure that our improvements are not simply due to chance, we compared the distribution of the cumulative runtimes of EP_0 and RW, HP and HELIOS as well EP_0 and HELIOS by using a Wilcoxon paired signed rank test at a significance level of 95%. On all datasets, all tests return significant results, which shows that the RW, HP and HELIOS lead to statistically significant runtime improvements.

5 Related Work

The task we address shares some similarities with the task of query optimization in databases [15]. A large spectrum of approaches have been devised to achieve this goal including System R's dynamic programming query optimization [13], cost-based optimizers and heuristic optimizers [6] and approaches based on genetic programming [1]. HELIOS is most tightly related to heuristic optimizers as it relies on an optimality assumption to discover plans in polynomial time. Overviews of existing approaches can be found in [2, 15]. The main difference between the task at hand and query optimization for databases are as follows: First, 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, HELIOS must rely on statistics that can be computed

efficiently while reading the data. Moreover, our approach 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.

This work is a contribution to the research area of LD. Several frameworks have been developed to achieve this goal. The LIMES framework [9], in which HELIOS is embedded, provides time-efficient algorithms for running specific atomic measures (e.g., PPJoin+ [17] and $\mathcal{HR}^3$ [7]) and combines them by using set operators and filters. While LIMES relied on graph traversal until now, most other systems rely on blocking. For example, SILK [5] relies on MultiBlock to execute LS efficiently. Multiblock allows mapping a whole link specification in a space that can be segmented to overlapping blocks. The similarity computations are then carried out within the blocks only. A similar approach is followed by the KnoFuss system [10]. Other time-efficient systems include [16] which present a lossy but time-efficient approach for the efficient processing of LS. Zhishi.links on the other hand relies on a pre-indexing of the resources to improve its runtime [11]. CODI uses a sampling-based approache to compute anchor alignments to reduce the its runtime [4]. Other systems descriptions can be found in the results of the Ontology Alignment Evaluation Initiative [3].[10] The idea of optimizing the runtime of schema matching has also been considered in literature [14]. For example, [12] presents an approach based on rewriting. Still, to the best of our knowledge, HELIOS is the first optimizer for link discovery that combines rewriting and planning to improve runtimes.

6 Conclusion and Future Work

We presented HELIOS, the (to the best of our knowledge) first execution optimizer for LS. We evaluated our approach in manually created and automatically generated LS. Our evaluation shows that HELIOS outperforms the canonical execution planner implemented in LIMES by up to two orders of magnitude. Our approach was intended to be generic. Thus, we used generic evaluation functions that allowed to detect plans that should generally work. Our results suggest that using more dataset-specific features should lead to even better runtimes and higher improvements. We thus regard HELIOS as the first step in a larger agenda towards creating a new generation of self-configuring and self-adapting LD frameworks. During the development of HELIOS, we noticed interesting differences in the behaviour of LD algorithms for different languages. For example, the Γ vector for the different measures differs noticeably for French, English and German. We will investigate the consequences of these differences in future work. Moreover, we will investigate more elaborate features for approximating the selectivity and runtime of different algorithms.

Acknowledgement

This work was partially financed the EU FP7 project GeoKnow (GA: 318159) and the DFG project LinkingLOD.

[10] http://ontologymatching.org

References

1. Kristin Bennett, Michael C. Ferris, and Yannis E. Ioannidis. A genetic algorithm for database query optimization. In *In Proceedings of the fourth International Conference on Genetic Algorithms*, pages 400–407, 1991.
2. Surajit 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.
3. Jérôme Euzenat, Alfio Ferrara, Willem Robert van Hage, Laura Hollink, Christian Meilicke, Andriy Nikolov, Dominique Ritze, François Scharffe, Pavel Shvaiko, Heiner Stuckenschmidt, Ondrej Sváb-Zamazal, and Cássia Trojahn dos Santos. Results of the ontology alignment evaluation initiative 2011. In *OM*, 2011.
4. Jakob Huber, Timo Sztyler, Jan Nößner, and Christian Meilicke. Codi: Combinatorial optimization for data integration: results for oaei 2011. In *OM*, 2011.
5. R. Isele, A. Jentzsch, and C. Bizer. Efficient Multidimensional Blocking for Link Discovery without losing Recall. In *WebDB*, 2011.
6. Carl-Christian Kanne and Guido Moerkotte. Histograms reloaded: the merits of bucket diversity. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 663–674. ACM, 2010.
7. Axel-Cyrille Ngonga Ngomo. Link discovery with guaranteed reduction ratio in affine spaces with minkowski measures. In *International Semantic Web Conference (1)*, pages 378–393, 2012.
8. Axel-Cyrille Ngonga Ngomo and Klaus Lyko. Eagle: Efficient active learning of link specifications using genetic programming. In *Proceedings of the Extende Semantic Web Conference*, pages 149–163, 2012.
9. Axel-Cyrille Ngonga Ngomo. On link discovery using a hybrid approach. *Journal on Data Semantics*, 1:203 – 217, December 2012.
10. Andriy Nikolov, Mathieu D'Aquin, and Enrico Motta. Unsupervised learning of data linking configuration. In *Proceedings of ESWC*, 2012.
11. Xing Niu, Shu Rong, Yunlong Zhang, and Haofen Wang. Zhishi.links results for oaei 2011. In *OM*, 2011.
12. Eric Peukert, Henrike Berthold, and Erhard Rahm. Rewrite techniques for performance optimization of schema matching processes. In *EDBT*, pages 453–464, 2010.
13. P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.
14. Pavel Shvaiko and Jérôme Euzenat. Ontology matching: State of the art and future challenges. *IEEE Trans. Knowl. Data Eng.*, 25(1):158–176, 2013.
15. Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.
16. Dezhao Song and Jeff Heflin. Automatically generating data linkages using a domain-independent candidate selection approach. In *ISWC*, pages 649–664, 2011.
17. Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.