Dynamic Planning for Link Discovery

Kleanthi Georgala[1✉], Daniel Obraczka[1], and Axel-Cyrille Ngonga Ngomo[2]

[1] AKSW Research Group, University of Leipzig, Augustusplatz 10, 04103 Leipzig
georgala@informatik.uni-leipzig.de
soz11ffe@studserv.uni-leipzig.de

[2] Paderborn University, Data Science Group, Pohlweg 51, D-33098 Paderborn, Germany
axel.ngonga@upb.de

Abstract. With the growth of the number and the size of RDF datasets comes an increasing need for scalable solutions to support the linking of resources. Most Link Discovery frameworks rely on complex link specifications for this purpose. We address the scalability of the execution of link specifications by presenting the first dynamic planning approach for Link Discovery dubbed Condor. In contrast to the state of the art, Condor can re-evaluate and reshape execution plans for link specifications during their execution. Thus, it achieves significantly better runtimes than existing planning solutions while retaining an F-measure of 100%. We quantify our improvement by evaluating our approach on 7 datasets and 700 link specifications. Our results suggest that Condor is up to 2 orders of magnitude faster than the state of the art and requires less than 0.1% of the total runtime of a given specification to generate the corresponding plan.

1 Introduction

The provision of links between knowledge bases is one of the core principles of Linked Data.[3] Hence, the growth of knowledge bases on the Linked Data Web in size and number has led to a significant body of work which addresses the two key challenges of Link Discovery (LD): efficiency and accuracy (see [1] for a survey). In this work, we focus on the first challenge, i.e., on the efficient computation of links between knowledge bases. Most LD frameworks use combinations of atomic similarity measures by means of *specification operators* and *thresholds* to compute link candidates. The combinations are often called linkage rules [2] or link specifications (short LSs, see Figure 1 for an example and Section 2 for a formal definition) to compute links [1]. So far, most approaches for improving the execution of LSs have focused on reducing the runtime of the atomic similarity measures used in LSs (see, e.g., [3,4,5]). While these algorithms have led to significant runtime improvements, they fail to exploit global knowledge about the LSs to be executed. In Condor, we *build upon these solutions* and tackle the problem of *executing link specifications efficiently*.

Condor makes used of a minute but significant change in the planning and execution of LSs. So far, the execution of LSs has been modeled as a linear process (see [1]), where a LS is commonly rewritten, planned and finally executed.[4] While this architecture has its merits, it fails to use a critical piece of information: *the execution engine*

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

[4] Note that some systems implement the rewriting and planning in an implicit manner.

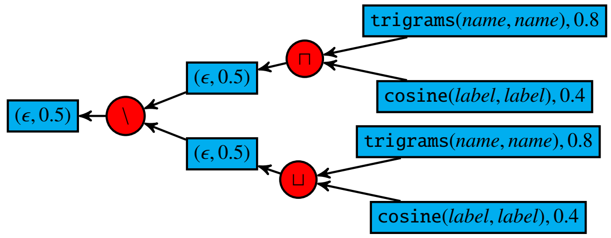


Fig. 1: Graphical representation of an example LS

knows more about runtimes than the planner once it has executed a portion of the specification. The core idea behind our work is to make use of the 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 (see Sect. 4).

The contributions of this work are hence as follows: (1) We propose the first planner for link specification which is able to re-plan steps of an input LS 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 LSs and 7 datasets and show that we outperfom the state of the art significantly.

2 Preliminaries

The formal framework underlying our preliminaries is derived from [6,7]. LD frameworks aim to compute the set $M = \{(s, t) \in S \times T : R(s, t)\}$ where S and T are sets of RDF resources and R is a binary relation. Given that M is generally difficult to compute directly, declarative LD frameworks compute an approximation $M' \subseteq S \times T \times \mathbb{R}$ of M by executing a *link specification* (LS), which we define formally in the following.

An *atomic LS L* is a pair $L = (m, \theta)$, where m is a similarity measure that compares properties of pairs (s, t) from $S \times T$ and θ is a similarity threshold. LS can be combined by means of operators and filters. Here, we consider the binary operators $\sqcup$, $\sqcap$ and $\backslash$, which stand for the union, intersection and difference of specifications respectively. *Filters* are pairs (f, τ), where f is either empty (denoted ϵ), a similarity measure or a combination of similarity measures and τ is a threshold.

A *complex LS L* is a triple $(f, \tau, \omega(L_1, L_2))$ where ω is a specification operator and (f, τ) is a filter. An example of a LS is given in Fig. 1. Note that an atomic specification can be regarded as a filter (f, τ, X) with $X = S \times T$. Thus we will use the same graphical representation for filters and atomic specifications. We call (f, τ) the *filter of L* and denote it with $\varphi(L)$. For our example, $\varphi(L) = (\epsilon, 0.5)$. The *operator of a LS L* will be

Table 1: Semantics of link specifications

L	$[[L]]$
(m, θ)	$\{(s, t, m(s, t)) \in S \times T : m(s, t) \geq \theta\}$
(f, τ, X)	$\begin{cases} \{(s, t, r) \in [[X]] : r \geq \tau\} \text{ if } f = \epsilon \\ \{(s, t, r) \in [[X]] : f(s, t) \geq \tau\} \text{ else.} \end{cases}$
$\sqcap(L_1, L_2)$	$\{(s, t, r) \mid (s, t, r_1) \in [[L_1]] \wedge (s, t, r_2) \in [[L_2]] \wedge r = \min(r_1, r_2)\}$
$\sqcup(L_1, L_2)$	$\left\{(s, t, r) \mid \begin{cases} r = r_1 \text{ if } \exists(s, t, r_1) \in [[L_1]] \wedge \neg(\exists r_2 : (s, t, r_2) \in [[L_2]]), \\ r = r_2 \text{ if } \exists(s, t, r_2) \in [[L_2]] \wedge \neg(\exists r_1 : (s, t, r_1) \in [[L_1]]), \\ r = \max(r_1, r_2) \text{ if } (s, t, r_1) \in [[L_1]] \wedge (s, t, r_2) \in [[L_2]]. \end{cases} \right\}$
$\backslash(L_1, L_2)$	$\{(s, t, r) \mid (s, t, r) \in [[L_1]] \wedge \neg\exists r' : (s, t, r') \in [[L_2]]\}$
$\emptyset(L)$	$[[L]]$

denoted $op(L)$. For $L = (f, \tau, \omega(L_1, L_2))$, $op(L) = \omega$. In our example the operator of the LS is $\backslash$. The *size of* L, denoted $|L|$, is defined as follows: If L is atomic, then $|L| = 1$. For complex LSs $L = (f, \tau, \omega(L_1, L_2))$, we set $L = |L_1| + |L_2| + 1$. The LS shown in Fig. 1 has a size of 7. For $L = (f, \tau, \omega(L_1, L_2))$, we call L_1 resp. L_2 the left resp. right direct child of L.

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*. We begin by assuming the natural semantics of the combinations of measures. The semantics of LSs are then as shown in Table 1. 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 architecture consisting of an execution engine, an optional rewriter and a planner (see [1] for more details). 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.[5] For the LS shown in Fig. 1, the execution plan returned by Canonical would thus first compute the mapping $M_1 = [[(\texttt{cosine}(label, label), 0.4)]]$ of pairs of resources whose property `label` has a cosine similarity equal to, or greater than 0.4. The computation of $M_2 = [[(\texttt{trigrams}(name, name), 0.8)]]$ would follow. Step 3 would be to compute $M_3 = M_1 \sqcup M_2$ while abiding by the semantics described in Table 1. Step 4 would be to filter the results by only keeping pairs that have a similarity above 0.5 and so on. 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 Condor

The goal of Condor 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 [7]) is to approximate the costs of possible

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

plans for L, and to simply select the least costly (i.e., the presumable fastest) plan so as 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 P, 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 generated afore 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*.

3.1 Planning

Algorithm 1 summarizes the dynamic planning approach implemented by CONDOR. The algorithm (dubbed *plan*) takes a LS L as input and returns the plan $P(L)$ with the smallest expected runtime. The core of the approach consists of (1) a cost function r which computes expected runtimes and (2) a recursive cost evaluation scheme. CONDOR's planner begins by checking whether the input L has already been executed within the current run (Line 2). If L has already been executed, there is no need to re-plan the LS. Instead, *plan* returns the known plan $P(L)$. If L has not yet been executed, we proceed by first checking whether L is atomic. If L is atomic, we return $P = run(m, \theta)$ (line 6), which simply computes $[[L]]$ on $S \times T$. Here, we make use of existing scalable solutions for computing such mappings [1].

If $L = (f, \tau, \omega(L_1, L_2))$, *plan* derives a plan for L_1 and L_2 (lines 10 and 11), then computes possible plans given $op(L)$ and 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* evaluates three alternative plans: (1) The *canonical* plan (lines 21, 23, 27, 31), which consists of executing $P(L_1)$ and $P(L_2)$, performing an intersection between the resulting mappings and then filtering the final mapping using (f, τ); (2) The *filter-right* plan (lines 24, 32), where the best plan P_1 for L_1 is executed, followed by a run of a filtering operation on the results of P_1 using $(f_2, \tau_2) = \varphi(L_2)$ and then filtering the final mapping using (f, τ); (3) The *filter-left* plan (lines 28, 32), which is a *filter-right* plan with the roles of L_1 and L_2 reversed.

As mentioned in Section 1, 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 ($r(P_1) = r(P_2) = 0$), 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 20). If P_1 resp. P_2 have already been executed (see Line 22 resp. 26), 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 Sect. 3.2 and the least costly plan is chosen. Similar approaches are implemented for $op(L) = \setminus$ (lines 12- 18). In particular, in line 17, 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 allowing only links of the retrieved mapping of L_1 that are not returned by L_2. For $op(L) = \sqcup$ (line 36) the plan always consists of merging the results of $P(L_1)$ and $P(L_2)$ by using the semantics described in Table 1.

3.2 Plan Evaluation

As explained in the first paragraphs of Sect. 2, one important component of Condor is the cost function required to estimate the costs of executing the corresponding plan. Based on [8], we used a linear plan evaluation schema as introduced in [7]. A plan P is characterized by one basic component, $r(P)$, the approximated runtime of executing P.

Approximation of $r(P)$ for atomic LSs. We compute $r(P(L))$ by assuming that the runtime of $L = f(m, \theta)$ can be approximated in linear time for each metric m using the following equation:

$$r(P) = \gamma_0 + \gamma_1|S| + \gamma_2|T| + \gamma_3\theta \ , \quad (1)$$

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. We used a linear model with these parameters since the experiments in [8] and [7] suggested that they are sufficient to produce accurate approximations. The next step of our plan evaluation approach was to estimate the parameters $\gamma_0, \gamma_1, \gamma_2$ and γ_3. However, the size of the source and the target KBs is unknown prior to the linking task. Therefore, we used a sampling method, where we generated source and target datasets of sizes $1000, 2000, \ldots, 10000$ by sampling data from the English labels of DBpedia 3.8. and stored the runtime of the measures implemented by our framework for different thresholds θ between 0.5 and 1. Then, we computed the γ_i parameters by deriving the solution of the problem to the linear regression solution of $\Gamma = (R^T R)^{-1} R^T Y$, where $\Gamma = (\gamma_0, \gamma_1, \gamma_2, \gamma_3)^T$, Y is a vector in which the y_i-th row corresponds to the runtime retrieved by running ith experiment and R is a four-column matrix in which the corresponding experimental parameters $(1, |S|, |T|, \theta)$ are stored in the r_i-th row.

Approximation of $r(P)$ for complex LSs. For the *canonical* plan, $r(P)$ is estimated by summing up the $r(P)$ of all plans that correspond to children specifications of the complex LS. For the *filter-right* and *filter-left* plans, $r(P)$ is estimated by summing the $r(P)$ of the child LS whose plan is going to be executed along with the approximation of the runtime of the filtering function 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) $r(P)$ includes only the sums of the children LSs that have not yet been executed. (2) If both children of the LS are executed then $r(P)$ 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.

3.3 Execution

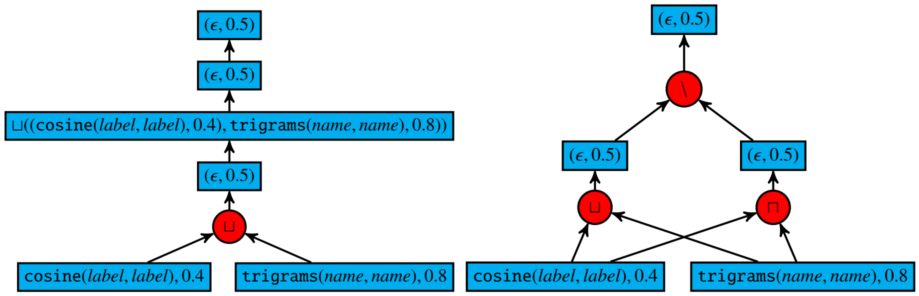
Algorithm 2 describes the execution of the plan that Algorithm 1 returned. The *execute* algorithm takes as input a LS L and returns the corresponding mapping M once all steps of $P(L)$ have been executed. The algorithm begins in line 2, where *execute* returns the mapping M of L, if L has already been executed and its result cached. If L has not been executed before, we proceed by checking whether a LS L' with $[[L]] \subseteq [[L]]'$ has already been executed (line 7). If such a L' exists, then *execute* retrieves $M' = [[L]]'$ and runs $(f, \tau, [[L]]')$ where $(f, \tau) = \varphi(L)$ (line 9). If $\nexists L'$, the algorithm checks whether L is atomic. If this is the case, then $P(L) = run(m, \theta)$ computes $[[L]]$. If $L = (f, \tau, \omega(L_1, L_2))$, *execute* calls the *plan* function described previously.

3.4 Example Run

To elucidate the workings of Condor further, we use the LS described in Fig. 1 as a running example. Table 2 shows the cost function $r(P)$ of 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 10, 11) 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. 3.1. Table 2 shows the approximation $r(P)$ of each plan for $(\sqcap((\texttt{cosine}(label, label), 0.4), (\texttt{trigrams}(name, name), 0.8)), 0.5)$. The least costly plan for the right child is the *filter-left* plan, where $L' = (\texttt{trigrams}(name, name), 0.8))$ is executed and $[[L']]$ is then filtered using $(\texttt{cosine}(label, label), 0.4))$ and $(\epsilon, 0.5)$. Before proceeding to discover the best plan for L, Condor assigns an approximate runtime $r(P)$ to each child plan of L: 3.5 s for the left child and 1.5 s for the right child.

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 executed previously, *plan* goes to line 15. There, it has to chose between two alternative plans, i.e., the *canonical* plan with $r(P) = 6.2$ s and the *filter-right* plan with $r(P) = 5.2$ s. It is obvious that *plan* is going to 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 Fig. 2a to $P(L)$ in line 14. For the next step, *execute* retrieves the left child $(\sqcup((\texttt{cosine}(label, label), 0.4), (\texttt{trigrams}(name, name), 0.8)), 0.5)$ and assigns it to L_1 (line 15). Then, the algorithm calls *execute* for L_1. *execute* repeats the plan procedure for L_1 recursively and returns the plan illustrated in Fig. 3. The plan is executed and finally (line 16) the resulting mapping is assigned to M_1. Remember that all intermediate mappings as well as the final mapping along with the corresponding LSs are stored for future use (line 29). Additionally, we replace the cost value estimations of each executed plan by their real values in line 28. Now, the cost value of $(\texttt{cosine}(label, label), 0.4)$ is assigned



(a) Initial plan (b) Final plan

Fig. 2: Initial and final plans returned by Condor for the LS in Fig. 1

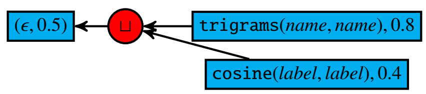


Fig. 3: Plan of the left child for the LS in Fig. 1

to 2.0 s, the cost value of $(\texttt{trigrams}(name, name), 0.8)$ is assigned to 1.0 s and finally, the cost value of the left child will be replaced by 4.0 s.

Now, given the runtimes from the execution engine, the algorithm re-plans the further steps of L. Within this second call of *plan* (line 17), Condor does not re-plan the sub-specification that corresponds to L_1, since its plan (Fig. 3) has been executed previously. Initially, *plan* had decided to use the right child as a filter. However, both $(\texttt{cosine}(label, label), 0.4)$ and $(\texttt{trigrams}(name, name), 0.8)$ 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 Fig. 2b.

Once the new plan $P(L)$ is constructed, *execute* checks if $P(L)$ includes any operators. In our example, $op(L) = \backslash$. Thus, we execute the second direct child of L as described in $P(L)$, $L_2 = (\sqcap((\texttt{cosine}(label, label), 0.4), (\texttt{trigrams}(name, name), 0.8)),$ $0.5)$. Algorithm 2 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. 3 by replacing the $\sqcup$ operator with the $\sqcap$ operator, with $r(P(L_2)) = 0$ s.

When the algorithm proceeds to executing $P(L_2)$, 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 $(\texttt{cosine}(label, label), 0.4)$ and $(\texttt{trigrams}(name, name), 0.8)$, filters the resulting mapping of the intersection with $(\epsilon, 0.5)$ and stores the resulting mapping for future use (line 29). 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 line 21 and 27. The final plan constructed by Condor is presented in Fig. 2b.

If the second call of the *plan* function for L in line 17 had resulted in not altering the initial $P(L)$, 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), 0.4), (trigrams(*name*, *name*), 0.8)), 0.5) (line 24). Similarly operations would have been carried out if $op(L) = \sqcap$ in line 26.

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 $O(|L|^2)$.

4 Evaluation

4.1 Experimental Setup

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? To address these questions, we evaluated our approach against seven data sets. The first four are the benchmark data sets for LD dubbed Abt-Buy, Amazon-Google Products, DBLP-ACM and DBLP-Scholar described in [9]. 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 created three additional data sets (MOVIES, TOWNS and VILLAGES, see Table 3) from the data sets DBpedia, LinkedGeodata and LinkedMDB. [6] [7] Table 3 describes their characteristics and presents the properties used when linking the retrieved resources. The mapping properties were provided to the link discovery algorithms underlying our results. We generated 100 LSs for each dataset by using the unsupervised version of Eagle, a genetic programming approach for learning LSs [10]. 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 return the same set of links and that they hence all achieve 100% F-measure w.r.t. the LS to be executed. [8]

4.2 Results

We compared the execution time of Condor with that of the state-of-the-art algorithm for planning (Helios [7]) as well as with the canonical planner implemented in Limes.

[6] http://www.linkedmdb.org/

[7] The new data and a description of how they were constructed are available at http://titan.informatik.uni-leipzig.de/kgeorgala/DATA/.

[8] 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

We chose LIMES because it is a state-of-the-art declarative framework for link discovery which ensures result completeness. Figure 4 shows the runtimes achieved by the different algorithm in different settings. As shown in Figure 4a, 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:

Answer to Q_1: CONDOR outperforms the state of the art in planning by being able to generate more time-efficient plans than HELIOS and CANONICAL.

Fig. 4a 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 the CANONICAL planner 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 approximately 600 s per specification on average. 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 4). 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 and outperforms planning and re-planning.

To answer Q_3, we also computed the runtime of LSs depending on their size t (see Figures 4b and 4c). 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 7.5% faster than the canonical plans on average. 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 $r(P)$ function. We can summarize our result as follows.

Answer to Q_3: CONDOR's performance gain over the state of the art grows with the size of the specifications.

Algorithm 1: *plan* Algorithm for CONDOR

```
Input: a link specification L;
Mapping of executed LS to plans specToPlanMap
Output: Least costly plan P of L
P ⟵ ∅
if specToPlanMap.contains(L) then
    P ⟵ specToPlanMap.get(L) //return plan stored in buffer for L
else
    if (L == (m, θ)) then
        P ⟵ run(m, θ)
    else
        L1 = L.leftChild
        L2 = L.rightChild
        P1 ⟵ plan(L1)
        P2 ⟵ plan(L2)
        if (L.operator == \) then
            if specToPlanMap.contains(L2) then
                P ⟵ merge(minus, P1, P2)
            else
                Q0 ⟵ merge(minus, P1, P2)
                Q1 ⟵ merge(getReverseFilter(φ(L2)), P1)
                P ⟵ getLeastCostly(Q0, Q1)
        else if (L.operator == ⊓) then
            if (specToPlanMap.contains(L1) ∧ specToPlanMap.contains(L2)) then
                P ⟵ merge(intersection, P1, P2)
            else if (specToPlanMap.contains(L1) ∧ ¬specToPlanMap.contains(L2))
              then
                Q0 ⟵ merge(intersection, P1, P2)
                Q1 ⟵ merge(φ(L2), P1)
                P ⟵ getLeastCostly(Q0, Q1)
            else if (¬specToPlanMap.contains(L1) ∧ specToPlanMap.contains(L2))
              then
                Q0 ⟵ merge(intersection, P1, P2)
                Q1 ⟵ merge(φ(L1), P2)
                P ⟵ getLeastCostly(Q0, Q1)
            else
                Q0 ⟵ merge(intersection, P1, P2)
                Q1 ⟵ merge(φ(L2), P1)
                Q2 ⟵ merge(φ(L1), P2)
                P ⟵ getLeastCostly(Q0, Q1, Q2)
        else
            P ⟵ merge(union, P1, P2) //last possible operator is ⊔
        specToPlanMap.put(L, P)
Return P
```

Algorithm 2: *execute* Algorithm

Input: a link specification L; mapping $specToPlanMap$; result buffer $results$

Output: Mapping M of L

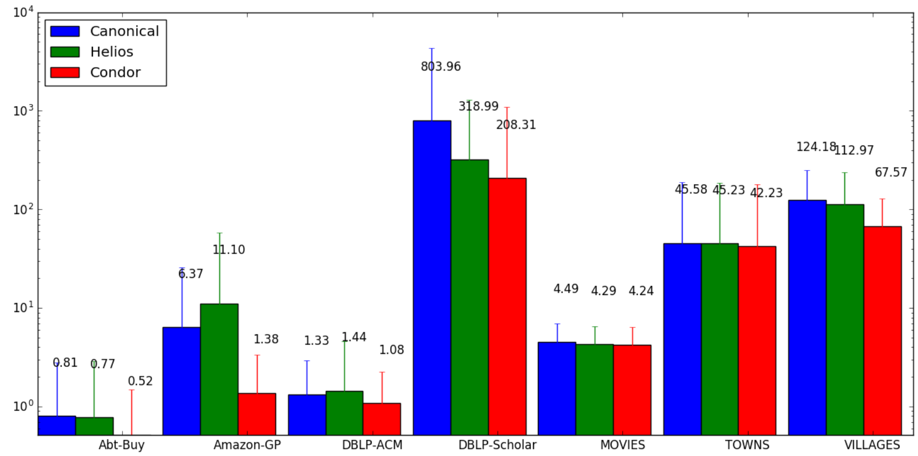
```
M ⟵ ∅
if (specToPlanMap.contains(L) then
    M ⟵ results.get(L)
    get the value for the key L
else
    L′ = checkDependencies(L, results)
    if L′ ≠ null then
        M′ ⟵ results.get(L′)
        M = filter(φ(L), M′)
    else
        if L = (m, θ) then
            M ⟵ run(m, θ)
        else
            P ⟵ plan(L)
            L1 ⟵ P.getSubSpec(0)
            M1 ⟵ execute(L1)
            P ⟵ plan(L)
            if op(P) ≠ ∅ then
                L2 ⟵ P(L).getSubSpec(1)
                M2 ⟵ execute(L2)
                M ⟵ runOperator(op(P), M1, M2)
            else
                if L = (f, τ, \(L1, L2)) then
                    M ⟵ filter(getReverseFilter(φ(L2)), M1)
                else
                    M ⟵ filter(φ(L2), M1)
            M ⟵ filter(φ(L), M)
    update()
    results ⟵ results.put(L, M)
Return M
```

Table 2: Runtime costs for the plans computed for the specification in (Fig. 1) by the two calls of the *plan* in lines 14 and 17. 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 executed previously ($^{\bullet}$) or have an estimation of zero cost in the current call of *plan* (*), (3) a runtime approximation value, that includes only runtimes of subsequent plans that have not been executed yet ($^{\square}$).

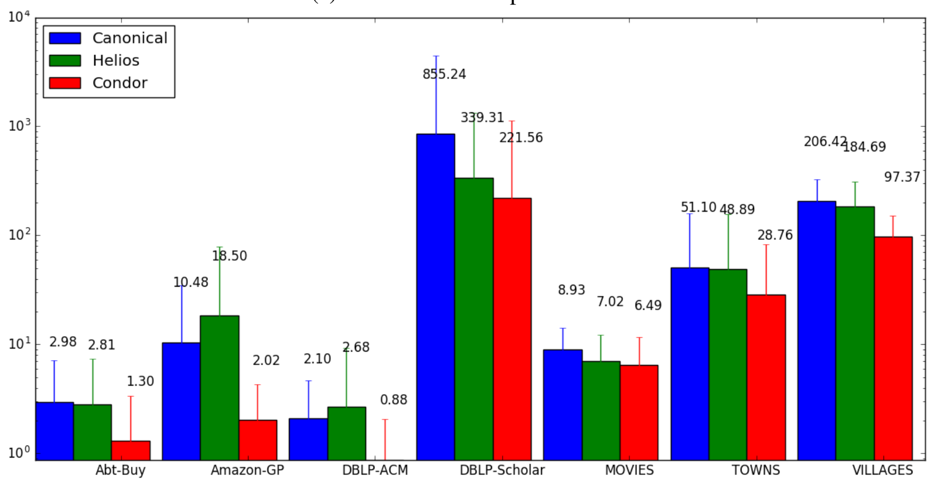
P	$r(P)$	
	1^{st}	2^{nd}
$(\texttt{cosine}(label, label), 0.4)$	1.8	$2.0^{\diamond}$
$(\texttt{trigrams}(name, name), 0.8)$	0.5	$1.0^{\diamond}$
$\varphi(\texttt{cosine}(label, label), 0.4)$	0.8	$0.8^{\square}$
$\varphi(\texttt{trigrams}(name, name), 0.8)$	0.6	$0.6^{\square}$
canonical plan: $merge(\sqcap, (\texttt{cosine}(label, label), 0.4), (\texttt{trigrams}(name, name), 0.8))$	3.5	$0.0^{\bullet}$
filter-right plan: $merge(\varphi(\texttt{trigrams}(name, name), 0.8), (\texttt{cosine}(label, label), 0.4))$	2.6	$0.8^{\square}$
filter-left plan: $merge(\varphi(\texttt{cosine}(label, label), 0.4), (\texttt{trigrams}(name, name), 0.8))$	1.5	$1.0^{\square}$
canonical plan: $merge(\sqcup, (\texttt{cosine}(label, label), 0.4), (\texttt{trigrams}(name, name), 0.8))$	3.5	$4.0^{\diamond}$
canonical plan for L	6.2	0.0^{*}
filter-right plan for L (see Fig. 2a)	5.2	$1.7^{\square}$

Table 3: Characteristics of data sets

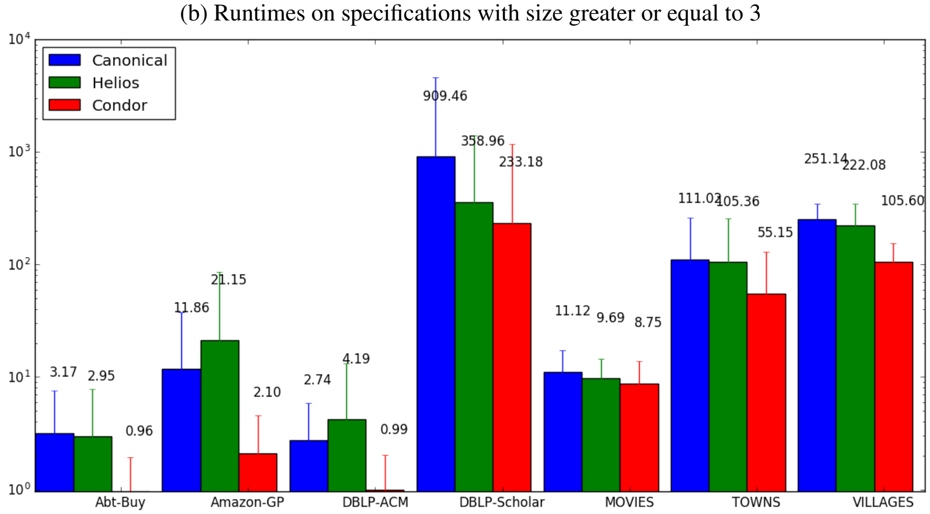
Data set	Source (S)	Target (T)	$\lvert S\rvert \times \lvert T\rvert$	Source Property	Target Property
Abt-Buy	Abt	Buy	1.20×10^6	product name, description manufacturer, price	product name, description manufacturer, price
Amazon-GP	Amazon	Google Products	4.40×10^6	product name, description manufacturer, 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
TOWNS	DBpedia	LGD	0.34×10^9	rdfs:label dbo:country/rdfs:label dbo:populationTotal geo:geometry	rdfs:label lgdo:isIn lgdo:population geom:geometry/agc:asWKT
VILLAGES	DBpedia	LGD	6.88×10^9	rdfs:label dbo:populationTotal geo:geometry	rdfs:label lgdo:population geom:geometry/agc:asWKT



(a) Runtimes on all specifications



(b) Runtimes on specifications with size greater or equal to 3



(c) Runtimes on specifications with size greater or equal to 5

Fig. 4: Mean and standard deviation of runtimes of CANONICAL, HELIOS and CONDOR. The y-axis shows runtimes in seconds on a logarithmic scale. The numbers on top of the bars are the average runtimes.

5 Related Work

This paper addresses the creation of better plans for scalable link discovery. A large number of frameworks such as SILK [2], LIMES [11] and KnoFuss [12] were developed to support the link discovery process. These frameworks commonly rely on scalable approaches for computing simple and complex specifications. For example, a lossless framework that uses blocking is *SILK* [2], a tool relying on rough index pre-matching. KnoFuss [12] on the other hand implements classical blocking approaches derived from databases. These approaches are not guaranteed to achieve result completeness. Zhishi.links [13] is another framework that scales (through an indexing-based approach) but is not guaranteed to retrieve all links. The completeness of results is guaranteed by the LIMES framework, which combines time-efficient algorithms such as Ed-Join and PPJoin+ with a set-theoretical combination strategy. The execution of LSs in LIMES is carried out by means of the CANONICAL [11] and HELIOS [7] planners. Given that LIMES was shown to outperform SILK in [7], we chose to compare our approach with LIMES. The survey of Nentwig et al. [1] and the results of the Ontology Alignment and Evaluation Initiative for 2017 of the OAEI [14],[9] provide an overview of further link discovery systems.

CONDOR is the first dynamic planner for link discovery. The problem we tackled in this work bears some resemblance to the task of query optimization in databases [15]. There have been numerous advances which pertain to addressing this question, including strategies based on genetic programming [16], cost-based and heuristic optimizers [17], and dynamic approaches [18]. Dynamic approaches for query planning were the inspiration for the work presented herein.

6 Conclusion and Future Work

We presented CONDOR, a dynamic planner for link discovery. We showed how our approach combines dynamic planning with subsumption and result caching to outperform the state of the art by up to two orders of magnitude. A large number of questions are unveiled by our results. First, our results suggest 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.

Acknowledgments

This work has been supported by H2020 projects SLIPO (GA no. 731581) and HOBBIT (GA no. 688227) as well as the DFG project LinkingLOD (project no. NG 105/3-2) and the BMWI Projects SAKE (project no. 01MD15006E) and GEISER (project no. 01MD16014)

[9] Ontology Alignment Evaluation Initiative: `http://ontologymatching.org`

References

1. Nentwig, M., Hartung, M., Ngomo, A.C.N., Rahm, E.: A Survey of Current Link Discovery Frameworks. Semantic Web (Preprint) (2015) 1–18
2. Isele, R., Jentzsch, A., Bizer, C.: Efficient Multidimensional Blocking for Link Discovery without losing Recall. In Marian, A., Vassalos, V., eds.: WebDB. (2011)
3. Ngonga Ngomo, A.C., Auer, S.: LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In: Proceedings of IJCAI. (2011)
4. Wang, J., Feng, J., Li, G.: Trie-join: Efficient Trie-based String Similarity Joins with Edit-distance Constraints. Proc. VLDB Endow. **3**(1-2) (September 2010) 1219–1230
5. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient Similarity Joins for Near Duplicate Detection. In: Proceedings of the 17th International Conference on World Wide Web. WWW '08, New York, NY, USA, ACM (2008) 131–140
6. Sherif, M.A., Ngomo, A.N., Lehmann, J.: Wombat - A Generalization Approach for Automatic Link Discovery. In: The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I. (2017) 103–119
7. Ngonga Ngomo, A.C.: HELIOS – Execution Optimization for Link Discovery. In: Proceedings of ISWC. (2014)
8. Georgala, K., Hoffmann, M., Ngomo, A.N.: An Evaluation of Models for Runtime Approximation in Link Discovery. In: Proceedings of the International Conference on Web Intelligence. WI '17, New York, NY, USA, ACM (2017) 57–64
9. Köpcke, H., Thor, A., Rahm, E.: Evaluation of Entity Resolution Approaches on Real-world Match Problems. Proc. VLDB Endow. **3**(1-2) (September 2010) 484–493
10. Ngonga Ngomo, A.C., Lyko, K. In: EAGLE: Efficient Active Learning of Link Specifications Using Genetic Programming. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 149–163
11. Ngonga Ngomo, A.C.: On Link Discovery using a Hybrid Approach. Journal on Data Semantics **1**(4) (Dec 2012) 203–217
12. Nikolov, A., d'Aquin, M., Motta, E.: Unsupervised Learning of Link Discovery Configuration. In: Proceedings of the 9th International Conference on The Semantic Web: Research and Applications. ESWC'12, Berlin, Heidelberg, Springer-Verlag (2012) 119–133
13. Niu, X., Rong, S., Zhang, Y., Wang, H.: Zhishi.links results for OAEI 2011. Ontology Matching (2011) 220
14. Achichi, M., Cheatham, M., Dragisic, Z., Euzenat, J., Faria, D., Ferrara, A., Flouris, G., Fundulaki, I., Harrow, I., Ivanova, V., Jiménez-Ruiz, E., Kuss, E., Lambrix, P., Leopold, H., Li, H., Meilicke, C., Montanelli, S., Pesquita, C., Saveta, T., Shvaiko, P., Splendiani, A., Stuckenschmidt, H., Todorov, K., Trojahn, C., Zamazal, O.: Results of the ontology alignment evaluation initiative 2016. In: OM 2016 : proceedings of the 11th International Workshop on Ontology Matching co-located with the 15th International Semantic Web Conference (ISWC 2016) Kobe, Japan, October 18, 2016. Volume 1766., Aachen, RWTH (2016) 73–129
15. Silberschatz, A., Korth, H., Sudarshan, S.: Database Systems Concepts. 5 edn. McGraw-Hill, Inc., New York, NY, USA (2006)
16. Bennett, K., Ferris, M.C., Ioannidis, Y.E.: A Genetic Algorithm for Database Query Optimization. In: Proceedings of the fourth International Conference on Genetic Algorithms. (1991) 400–407
17. Kanne, C.C., Moerkotte, G.: Histograms Reloaded: The Merits of Bucket Diversity. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD '10, New York, NY, USA, ACM (2010) 663–674
18. Ng, K.W., Wang, Z., Muntz, R.R., Nittel, S.: Dynamic query re-optimization. In: Scientific and Statistical Database Management, 1999. Eleventh International Conference on, IEEE (1999) 264–273