The Lazy Traveling Salesman – Memory Management for Large-Scale Link Discovery

Axel-Cyrille Ngonga Ngomo(✉) and Mofeed M. Hassan

AKSW Research Group, University of Leipzig,
Augustusplatz 10, 04103 Leipzig, Germany
{ngonga,mounir}@informatik.uni-leipzig.de
http://limes.sf.net

Abstract. Links between knowledge bases build the backbone of the Linked Data Web. In previous works, several time-efficient algorithms have been developed for computing links between knowledge bases. Most of these approaches rely on comparing resource properties based on similarity or distance functions as well as combinations thereof. However, these approaches pay little attention to the fact that very large datasets cannot be held in the main memory of most computing devices. In this paper, we present a generic memory management for Link Discovery. We show that the problem at hand is a variation of the traveling salesman problem and is thus NP-complete. We thus provide efficient graph-based algorithms that allow scheduling link discovery tasks efficiently. Our evaluation on real data shows that our approach allows computing links between large amounts of resources efficiently.

1 Introduction

A wide variety of data publishers are now making Linked Data available.[1] With this variety come millions of resources that need to be linked together to generate real five-star Linked Data. While one can rely on hardware architectures such as cloud-based processing to link large knowledge bases, previous works have shown that the bottleneck of having to transfer local data to the Cloud makes cloud-based solutions rather unattractive [13]. Local solutions on the other hand only provide limited amounts of memory that must commonly be shared with other processes. This in turn means that efficient memory management approaches for Link Discovery (LD) are needed to enable LD on large datasets while relying on local hardware. While the time complexity of LD has been studied considerably over the last years (see [10] for a survey), LD's space complexity has not been paid much attention to. In this paper, we address exactly this research gap and investigate a *generic memory management approach* for LD.

The rationale behind our approach, Gnome, is to allow deploying the paradigms commonly used for efficient LD (e.g., PPJoin+ [22], MultiBlock [9] and $\mathcal{HR}^3$ [11]) on very large datasets that do not fit in the main memory of the

[1] See, e.g., http://lod-cloud.net/ and http://stats.lod2.eu.

H. Sack et al. (Eds.): ESWC 2016, LNCS 9678, pp. 423–438, 2016.
DOI: 10.1007/978-3-319-34129-3_26

computer at hand. We implement this vision through the following research contributions: (1) We introduce best-effort and greedy graph-based algorithms for determining how a LD problem should be addressed to ensure good memory usage. (2) We justify the use of best-effort and greedy algorithms by mapping the problem of loading data into the memory in the right order to the traveling salesman problem (TSP), which is known to be NP-complete. (3) We show that while our algorithms are not optimal, they can deal with large datasets efficiently. (4) We provide empirical evidence to substantiate the use of greedy and best-effort approaches. To this end, we evaluate our approach on large real datasets derived from DBpedia and LinkedGeoData.

The rest of this paper is structured as follows: we present a formal specification of the problem at hand in Sect. 2. Then, we present the formal model behind GNOME in Sect. 3. In Sect. 4, we present GNOME in detail. We first give an overview of the intuitions behind the approach. In particular, we map the memory management problem to an edge-partitioning problem in combination with the TSP. The evaluation of GNOME and its components on datasets of different sizes is presented in Sect. 5. Here, we show that we can perform large-scale link discovery on commodity hardware. Finally, we give a brief overview of related work (Sect. 6) and conclude.

2 Preliminaries

The LD problem can be formalized as follows: Given a two sets of RDF resources S and T as well as a relation R, compute the set $M = \{(s,t) \in S \times T : R(s,t)\}$. Declarative LD frameworks usually compute an approximation $M' = \{(s,t) \in S \times T : \sigma(s,t) \geq \theta\}$ of M, where σ is a (complex) similarity function and θ is a similarity threshold. Naïve approaches to computing M' have a quadratic time complexity and at least a linear space complexity. The quadratic time complexity is due to the need to execute σ on all elements of $S \times T$. The linear space complexity comes from the need to load both S and T into the main memory of the computing device at hand. Over the last years, time-efficient approaches have been developed with the aim of reducing the runtime of declarative link discovery frameworks. One key insights that underlies GNOME is that while these approaches are based on varied models such as sequences of filters (e.g., PPJoin+ [22]), blocking (e.g., MultiBlock [9]) and space tiling (e.g., $\mathcal{HR}^3$ [11]) they all reduce the overall time complexity of LD by reducing the number of comparisons that need to be carried out to determine M' completely. This reduction is commonly achieved by determining automatically which (possibly overlapping) subsets of S must be compared with which (possibly overlapping) subsets of T to compute M' fully. We call approaches that abide by this model *divide-and-merge* LD approaches.

3 A Task Model for Efficient Link Discovery

Formally, our insight pertaining to divide-and-merge approaches translates into these approaches operating as follows: Given S, T, σ and θ (as defined above), they determine

1. the set $\mathcal{S}$ of subsets $S_1, \ldots, S_n$ of S, the set $\mathcal{T}$ of subsets $T_1, \ldots, T_m$ of T and
2. the mapping function $\mu : \mathcal{S} \to 2^{\mathcal{T}}$ such that
3. the elements of each S_i must only be compared with the elements of all sets in $\mu(S_i)$ and
4. the union of the results over all $S_i \in \mathcal{S}$ is exactly M', i.e., $M' = \{(s,t) : s \in S_i \wedge t \in T_j \wedge T_j \in \mu(S_i) \wedge \sigma(s,t) \geq \theta\} = \{(s,t) \in S \times T : \sigma(s,t) \geq \theta\}$.

We can thus model the computation of M' as the execution of a sequence of tasks E_{ij}, where each task E_{ij} consists of comparing all elements of S_i with all elements of T_j. We denote the set of all tasks with $\mathcal{E}$. We will regard the main memory of a computing device as a storage solution C with a limited capacity $|C|$ and three simple operators: `load (D)`, which loads the data item D into C, `evict(D)` which deletes D from C and `get(D)` which (1) returns D if C contains D or (2) evicts as many elements from C as necessary to be able to load D into C and to return it. If $|S| + |T| \leq |C|$, then the computation of M' can be carried in the main memory of the device at hand. In this work, we will be concerned with cases where S and T cannot be held in C, i.e., $|S| + |T| > |C|$. In this case, if S_i or T_j is not available in C, then E_{ij} can only be computed once the missing items are loaded into C. In many cases, this will require deleting some of the sets S'_i and T'_j that are already in C. Determining which sets are to be evicted from a storage solution is commonly solved by relying on *caching strategies*.

The main goal of this paper is correspondingly to *determine the right order for executing tasks against C so as to maximize the locality of data*, i.e., to minimize the amount of data transferred between the hard drive and the main memory. Our first intuition pertaining to this goal is that *μ provides hints towards which tasks should form a subsequence* of the sequence of tasks. We implement this insight by clustering tasks and carrying the tasks within a cluster after each other. Our second intuition is that *certain clusters rely on overlapping data*. Clusters that share a large amount of data should be executed after one another to improve the overall locality. We implement this insight by providing a best-effort solution to the corresponding TSP.

Throughout this paper, we assume that $\forall S_i \; \forall T_j \in \mu(S_i) : |S_i| + |T_j| \leq |C|$. We say that C is *sufficient* to compare S with T if it fulfills this condition. The sufficiency of C is necessary to ensure that all elements of S_i can be compared with all elements of all $T_j \in \mu(S_i)$. Note that mappings μ that do not abide by this condition can be extended to mappings that do abide by this restriction simply by partitioning S_i and T_j into smaller datasets and updating μ correspondingly.

4 Approach

This section presents our approach to memory management formally. We assume that we are given the sets $\mathcal{S}$ and $\mathcal{T}$, the mapping function μ and a fixed and sufficient amount of main memory C with size $|C|$. We begin by defining the formal concepts necessary to understand our approach. Then, we present graph-clustering-based approaches that allow improving the usage of C while also improving the data locality of the problem at hand. Finally, we show that determining the right sequence for processing clusters is NP-complete and present best-effort approaches to approximate this sequence.

4.1 From Tasks to Task Graphs

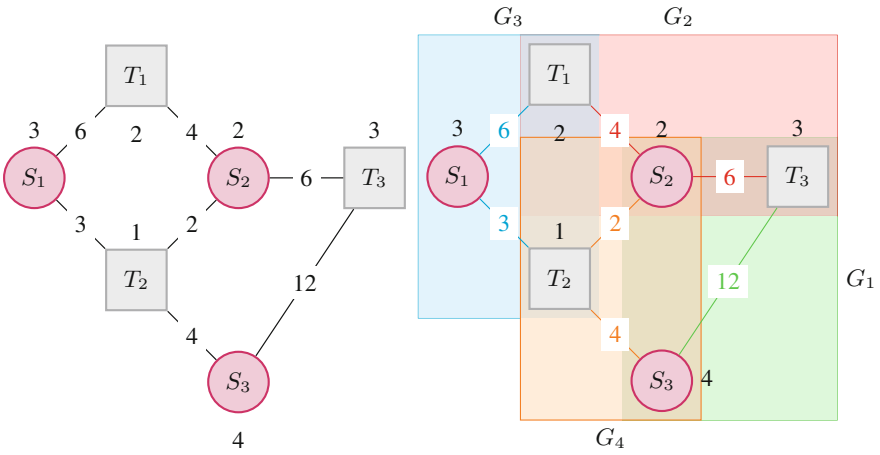
Our approach begins by modeling the tasks to a given LD problem as an undirected weighted graph $G = (V, E, w_v, w_e)$, which we call a *task graph*. The set V of vertices of G is the set $\mathcal{S} \cup \mathcal{T}$, which we call the set of *data items*. Two data items $S_i, T_j \in V$ are related by an edge $e = \{S_i, T_j\}$ iff $T_j \in \mu(S_i)$. The weight function $w_v : V \to \mathbb{N}$ maps each $v \in V$ to the total amount of main memory required to store it, i.e., $w_v(v) = |v|$. The weight function $w_e : V \to \mathbb{E}$ maps each $e = \{S_i, T_j\}$ to $w_v(e) = |S_i||T_j|$, i.e., to the total number of comparisons needed to process $S_i \times T_j$. Note that the edge $\{S_i, T_j\}$ corresponds exactly to the task E_{ij}. As a running example, we will consider a case where (1) $|C| = 7$, (2) S is subdivided into S_1, S_2 and S_3, (3) T is divided into T_1, T_2 and T_3. The corresponding task graph with sizes for the S_i and T_j is shown in Fig. 1a. For the sake of simplicity, we will use $V(X)$ resp. $E(X)$ in this paper to denote the set of vertices resp. of edges of a graph X.

Within this model, the insights behind our approach can be translated as follows: Maximizing the locality of data in C corresponds to finding the sequence of tasks that minimizes the total volume of the `load` operations by C. We address this problem in two steps. First, we devise several approaches to clustering the graph G so as to find clusters of tasks that should form a subsequence of the sequence of tasks to execute. Then, we approximate the order in which these clusters of tasks should be executed.

4.2 Clustering Tasks

The aim of clustering tasks is to detect portions of the graphs that describe groups tasks which should be executed after each other with the aim of ensuring high data locality. Formally, clustering is equivalent to detecting subsequences of the sequence of all tasks. We developed two types of approaches to achieve this goal: a naïve approach, which makes use of the locality of subsets of S and a greedy approach which aims to maximize an objective function efficiently.

Naïve Approach. A naïve yet time-efficient approach towards clustering lies in making use of G being a bi-partite graph by virtue of subsets of S being



(a) Task dependency graph.

(b) Task dependency graph clustered by edge weight.

Fig. 1. Example task graph. The nodes from S are circles while the nodes from T are rectangles. The weights of nodes are displayed next to the nodes. The weights of the edges are displayed above edges.

linked to subsets T exclusively. Hence, we can cluster G by simply creating a cluster $G_i = G(S_i)$ for each S_i with (1) $V(G(S_i)) = \{S_i\} \cup \mu(S_i)$ and (2) $E(G(S_i)) = \{e \in E(G) : e \subseteq V(G(S_i))\}$. For our example, we get a.o. $E(G_1) = E(G(S_1)) = \{\{S_1, T_1\}, \{S_1, T_2\}\}$ while $V(G_1) = \{S_1, T_1, T_2\}$. Note that the result of this approach is complete as it is guaranteed to cover all edges in G. The main advantage of the naïve approach is that is very time-efficient with a worst-case time complexity of $O(|\mathcal{E}|)$. However, it is clearly suboptimal. We thus also propose a clustering approach which generates solutions which have a higher locality.

Greedy Approach. Our more intricate approach towards clustering tasks takes $|C|$ into consideration by aiming to discover the portions of G that can fit in memory simultaneously while maximizing the number of comparisons that can be carried out while having this data in memory. Based on the graph model above, this aim translates to aiming to find a *division* of G into connected graphs $G_k = (V_k, E_k)$ that

1. are *node-maximal* w.r.t $|C|$, i.e., such that $\sum_{v \in V_k} |v| \leq |C|$ but building a connected graph $G'_k = (V'_k, E'_k)$ with $V_k \subset V'_k$ would lead to a graph with $\sum_{v \in V'_k} |v| > |C|$
2. are *complete*, i.e., $\bigcup_k V_k = V$ and $\bigcup_k E_k = E$
3. implement an *edge partitioning* of G, i.e., $\forall e \in E(G)\ \exists E_k : e \in E_k$ and $k \neq k' \rightarrow E_k \cap E_{k'} = \emptyset$.

Algorithm 1. Greedy Task Clustering Algorithm

```
   Input: Task graph G, |C|
 1 List L = sortAscending(E(G));
 2 Cluster G_k;
 3 clusters = ∅;
 4 while |L| > 0 do
 5     List candidateEdges = ∅;
 6     Cluster G_k = ∅;
 7     Edge e = L.firstElement();
 8     G_k.addEdge(e);
 9     L.remove(e);
10     candidateEdges.add(e.getRelatedEdges());
11     candidateEdges = sortAscending(candidateEdges);
12     counter = 0;
13     while ++counter < candidateEdges.size() do
14         e = candidateEdges.get(counter);
15         if (canBeAdded(G_k, e)) then
16             candidateEdges.remove(e);
17             L.remove(e);
18             G_k.addEdge(e);
19             candidateEdges.add(e.getRelatedEdges() ∩L);
20             candidateEdges = sortAscending(candidateEdges);
21             counter = 0;
22     clusters.add(G_k);
23 return clusters;
```

The nodes of each G_k are the data items that should be in memory at the same time while the edges of G_k allow deriving which tasks should be carried out when G_k is in memory. Hence, determining an appropriate division of G can be carried out as shown in Algorithm 1. We begin by sorting the edges in E by weight (see Line 11). Then, we create a new element G_k of the graph division. We select the unassigned edge e with the largest weight and add it to E_k while the corresponding nodes are added to V_k (see Line 7). e is then removed from the set of edges to assign (see Line 9). All unassigned edges that are reachable from V_k are then added to the set of candidate edges. We then pick the edge with the highest weight from this set of candidates, add it to E_k and add the corresponding nodes to V_k if does not break any of the conditions above (Lines 10–11). This procedure is repeated (see Line 13) until no edge can be added to G_k without breaking the node-maximality condition. We create new connected graphs G_k until all edges are covered, thus ensuring the completeness of the results generated by our approach.

For our running example, the approach begins with E_{33} as it has a size of 12. The edges E_{23} and E_{32} are then added to the list of candidates. Moreover, E_{33} is removed from the list of edges to process. As E_{23} nor E_{32} can be added without going against the maximality condition ($|S_3| + |T_3| = 7 = |C|$), the first

cluster G_1 is considered completed. The next cluster is started with E_{11}, which is one of the edges with the highest weights. Upon completion, our approach returns the clusters shown in Fig. 1b.

4.3 Scheduling Clusters of Tasks

While the approaches above allows deriving how to make good use of C locally (i.e., at the level of subsequences), they do not allow determining the order in which the clusters should be loaded into C to minimize the traffic between the hard drive and main memory. Our main insight towards addressing this problem is that all tasks can be interpreted as a weighted undirected graph H with nodes G_k and edge weights $w(\{G_k, G_{k'}\}) = \sum_{v \in V_k \cap V_{k'}} |v|$. Thus, finding a good sequence of nodes G_k is equivalent to finding a path in H that covers all nodes of H and bears a large weight, as a large overall weight signifies that several data items can be reused. Finding this path is equivalent solving a TSP where the total distance is to be maximized. We dub this variation of TSP the lazy TSP, as our salesman wants to stay on the road as much as possible to rest. The TSP being known to be NP-complete, we will not attempt to derive an optimal solution for the scheduling of tasks. Instead, we choose to use a best-effort approach.

Best-Effort Solution. We introduce the cluster overlap function, which measures the total weights of the nodes that two clusters have in common, i.e.,

$$o(G_i, G_j) = \sum_{v \in V(G_i) \cap V(G_j)} |v|. \tag{1}$$

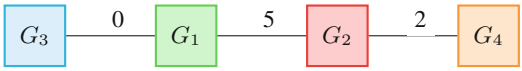
For our example shown in Fig. 1b, the overlap between G_1 and G_2 is 5 as they share the nodes S_2 and T_3. We extend this overlap function to sequences as follows:

$$o(G_1, \ldots, G_N) = \sum_{i=1}^{N-1} o(G_i, G_{i+1}). \tag{2}$$

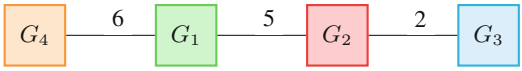
For example, the overlap of the sequence of clusters shown in Fig. 2a is 7. The basic idea behind our best-effort solution is to begin by the clusters set in a random order $G_1, \ldots, G_N$ and to find new permutations of the graphs G_i which improve the overlap function. The scalability of this approach is based on the following observation: After the permutation of the clusters with indexes i and j from a given sequence $G_1, \ldots, G_N$, the difference Δ in overlap is given by

$$\begin{aligned} \Delta = (&o(G_{i-1}, G_i) + o(G_i, G_{i+1}) + o(G_{j-1}, G_j) + o(G_j, G_{j+1})) - \\ &(o(G_{i-1}, G_j) + o(G_j, G_{i+1}) + o(G_{j-1}, G_i) + o(G_i, G_{j+1})). \end{aligned} \tag{3}$$

If $\Delta > 0$, then a sequence that leads to a higher locality was found. Note that this approach only requires computing 8 overlap scores for each new permutation and thus scales well.



(a) Initial sequence of clusters. Overlap = 7.



(b) Improved sequence of clusters. Overlap = 13.

Fig. 2. Examples of sequences of clusters. The overlap of pairs of clusters are displayed above the corresponding edges.

Algorithm 2. Overview of GNOME

```
   Input: Cache C, set 𝒮, set 𝒯, function μ
1  Graph G = computeTaskGraph(𝒮, 𝒯, μ);
2  List L = cluster(G);
3  L = schedule (L);
4  Mapping M = ∅;
5  for G_k ∈ L do
6      for {S_i, T_j} ∈ E_k do
7          source = C.get(S_i);
8          target = C.get(T_j);
9          M = M ∪ compare(source, target);
10 return M;
```

Our best-effort approach makes use of this insight by beginning with a random sequence. Then, it selects two indexes i and j at random and computes Δ for the sequence that would occur if G_i and G_j were swapped in the sequence. If $\Delta > 0$, then i and j are swapped. We iterate this procedure until a condition (e.g., a maximal runtime) is achieved and return the current best sequence found by the approach. In the example derived from Fig. 1b shown in Fig. 2a , let $i = 1$ and $j = 4$. This lead to G_3 (which has the index $i = 1$ in the sequence) being swapped with G_4 (index $j = 4$). The permutation (see Fig. 2b) leads to an increase of the overlap score to 13. Hence, it is kept by the approach. The main advantage of using such an approach is scalability as the overall overlap function does not need to be computed given that we can limit ourselves to finding permutations which improve the overlap. Hence, the approach can be set to run for a predefined amount of time or iterations (we hence dub it a best-effort approach) on any input dataset. For a fixed number of iterations I, the complexity of the approach is $O(I)$ as it always computes exactly $8I$ overlap scores per iteration.

Greedy Approach. For the sake of comparison, we also implemented a greedy approach. It starts with the first node within the input and finds a path that

covers all clusters G_i by always iteratively choosing the next node with the highest overlap to the current node that does not yet belong to the path. For our example, the approach would return the path (G_3, G_2, G_1, G_4) with an overlap score of 13. Note that while this approach is global and can return results with a higher overlap score than the best-effort approach, it has a worst-case complexity of $O(N^2)$, where N is the number of clusters.

Algorithm 2 summarizes our approach and shows how all the components interact. Given the sets $\mathcal{S}$ and $\mathcal{T}$ as well as the function μ, we begin by computing the corresponding task graph. The graph is then clustered. The generated clusters are forwarded to the scheduling approach, which determine the sequence in which the clusters are to be executed. Finally, the clusters are executed in the order given by the scheduler.

5 Evaluation

5.1 Experimental Setup

Within our experiments, we aimed to answer the following questions:

Q1 *How do the alternative configurations of our approach perform?*
Q2 *How does our approaches perform against existing caching strategies?*
Q3 *How well does our approach scale?*

To answer these questions, we evaluated our approach against baseline approaches on two real datasets derived from DBpedia and LinkedGeoData.[2] We only considered deduplication experiments, where $S = T$.

Datasets. We selected the first 1 million labels from DBpedia version 04-2015[3] as our first dataset DBP. To compare these labels, we used the trigram similarity as similarity measure [22]. We computed the sets S_i by first computing all trigrams found in labels in S. Then, each S_i was set to be the set of all resources (1) whose labels had a particular length and (2) whose labels contained a particular trigram. Note that this means that $\exists i, j : i \neq j \wedge S_i \cap S_j \neq \emptyset$. Let $n(S_i)$ resp. $tri(S_i)$ be length of the strings in S_i resp. the trigram they all contain. To compute $\mu(S_i)$, we selected all S_j which (1) contained strings of length between $n(S_i) \times \theta$ and $n(S_i)/\theta$ (where θ is the similarity threshold) and (2) contained $tri(S_i)$. These equations were derived from [22] and guarantee the completeness of our results.

Our second dataset (LGD) contained 800,000 places from LinkedGeoData as well as their latitude and longitude. The computation of the S_i and $\mu(S_i)$ was based entirely on $\mathcal{HR}^3$ [11]. Let τ be the distance threshold corresponding to the similarity threshold θ (i.e., $\frac{1}{1+\theta}$) expressed in degrees. Each S_i was the region

[2] The datasets and the corresponding result files are available at https://github.com/AKSW/LIMES/tree/master/LazyTravelingSalesMan.
[3] http://dbpedia.org/.

which contained resources with latitudes in $\left[\frac{k\tau}{4}, \frac{(k+1)\tau}{4}\right[$ and with longitudes in $\left[\frac{k'\tau}{4}, \frac{(k'+1)\tau}{4}\right[$ with $(k, k') \in \mathbb{Z}^2$. The set $\mu(S_i)$ contained all S_j that surrounded S_i and could contain points such that the euclidean distance between any point of S_j and of S_i could be less than τ. Note that in contrast to the approach used on DBpedia, the S_i here did not overlap.

Hardware. All experiments were carried out on a Linux Server running *OpenJDK* 64-Bit Server 1.8.0.66 on Ubuntu 14.04.3 LTS on Intel(R) Xeon(R) E5-2650 v3 processors clocked at 2.30 GHz. Each experiment was repeated three times and allocated 10 G of memory and 1 core to emulate good commodity hardware.

Measures. We measured the *runtime* of each approach by subtracting the current system time at the start of each approach from the current system time at the end of the same execution. The *number of hits* and misses was measured as follows: Each time that a data item D was required for computing a task E_{ij}, the required data was requested from the cache. If the data was found in the cache (cache hit), we added $|D|$ to the hit count $hits$, which was initialized with 0. If the data was not found in the cache, we proceeded accordingly with the number of misses $misses$. We report the *hit ratio* given by $hit\ ratio = hits/(hits+misses)$.

Baseline. Different existing caching strategies can be used to improve the memory management by reducing the time necessary to transfer data from the hard drive. We implemented five caching strategies with different characteristics and used them as baseline as well as in combination with our approach. These strategies are: (1) **FIFO** (First-In First-Out): Once a cache following this strategy is full, it evicts the element that stayed the longest in the cache. (2) **FIFO2ndChance** (First-In First-Out Second Chance): This cache modifies the FIFO strategy by evicting the recording elements that have let to hits and giving them a second chance if they are to be evicted by reinserting them into the cache as new entries. The oldest element in the cache that has not led to a hit is then removed. (3) **LRU** (Least Recently Used): Here, the entry that has been referenced the furthest in the past is evicted. (4) **LFU** (Least Frequently Used): The evicted element is the one with the smallest number of references (hits). (5) **SLRU** (Segmented Least Recently Used): an extension of LRU where the cache has two segments, a protected and unprotected segment. First, new elements are cached in the unprotected segment. Once the cache recorded a hit for such element, it is transfered to the protected segment. Eviction occurs in both segments when they are full. While eviction out of whole cache occurs to the elements in the unprotected segment, the elements in protected segment are transfered to the unprotected one.

5.2 Evaluation of Clustering

We compared the two clustering approaches developed herein w.r.t. their runtime and their hit ratio. In each experiment, we used the first 1,000 resp.

Table 1. Average clustering results on LinkedGeoData and DBpedia. The results for 1,000 resp. 10,000 resources are shown in the top resp. bottom section of the table.

$\|C\|$	Runtimes (ms)				Hit ratio			
	LGD		DBP		LGD		DBP	
	Naive	*Greedy*	*Naive*	*Greedy*	*Naive*	*Greedy*	*Naive*	*Greedy*
100	568.0	646.3	888.0	31,973.7	0.57	0.77	0.50	0.54
200	518.3	594.0	937.0	32,563.0	0.66	0.80	0.50	0.54
400	532.0	593.3	9,014.0	32,180.7	0.67	0.80	0.50	0.54
1,000	5,974.0	118,454.7	9,014.0	1,991,841.7	0.51	0.64	0.49	0.57
2,000	6,168.0	115,450.0	10,018.0	1,848,703.0	0.51	0.63	0.50	0.57
4,000	7,118.3	121,901.7	11,001.7	1,947,228.7	0.50	0.63	0.50	0.57

10,000 resources described in the datasets at hand. As caching approach, we used the FIFO strategy. Our results on LGD and DBP are shown in Table 1. The table shows clearly that while the greedy and naïve approach achieve similar runtimes on the LinkedGeoData fragment with 1,000 resources, the greedy clustering approach is orders of magnitude slower than the naïve approach in all other cases. Still, the results also show that a better clustering of tasks as performed by greedy clustering leads to higher hit ratios, thus suggesting that clustering alone can already be beneficial for improving the scheduling of link discovery tasks. Overall, we opted to use the naïve approach in all subsequent experiments.

5.3 Evaluation of Scheduling

We compared the scheduling approaches using the same setting as for the clustering. The runtime for the naïve scheduler was set to 250 ms. The results shown in Table 2 suggest similar insights as with clustering (the evaluation of LinkedGeoData led to similar results). While the more complex approach followed by the greedy scheduler leads to more hits (e.g., 68.6 % more hits on 10,000 resources on LGD), the total runtime that it requires makes it unusable for large datasets. Hence, we chose to use the best-effort approach with a threshold set to 250 ms for the rest of our experiments.

5.4 Combination of GNOME with Existing Caching Approaches

After completing the evaluation of the components of GNOME, we aimed to determine the caching approach with which GNOME performed best. To achieve this goal, we compared the run times and the number of hits achieved by the five caching approaches presented at the beginning of this section on the same data as in the precedent section. The results of our evaluation are shown in Table 3. While the number of hits achieved with the different approaches varies only

Table 2. Average scheduling results on LGD and DBP. The results for 1,000 resp. 10,000 resources are shown in the top resp. bottom section. BE stands for best-effort.

$\|C\|$	Runtimes (ms)				Hit ratio			
	LGD		DBP		LGD		DBP	
	BE	*Greedy*	*BE*	*Greedy*	*BE*	*Greedy*	*BE*	*Greedy*
100	571.3	1,599.3	887.0	64288.7	0.56	0.68	0.50	0.65
200	565.7	1,448.3	860.3	62305	0.66	0.85	0.50	0.65
400	581.0	1,379.3	918.7	60458.7	0.67	0.88	0.50	0.65
1,000	5,666.0	814,271.7	8825.7	3851388.3	0.51	0.86	0.49	0.73
2,000	6,268.0	810,855.0	9347	3,385,014.3	0.51	0.86	0.50	0.73
4,000	6,675.7	814,041.7	10,666.7	3364521.3	0.50	0.86	0.50	0.73

slightly, the run times achieved by our approach when combined with FIFO are clearly less than those achieved with any other method. These results confirm that our combination of clustering and path ensures a high locality of the data independently of the caching approach used. The difference in runtime is due to the partly complex operations that have to be performed by the cache to detect the data item(s) that is (are) to be evicted when a cache miss comes about and the cache is full. The FIFO strategy being simple means that the cache itself leads to a small overhead, leading to a smaller computation time. Given that these observations hold on both datasets, we opted to combine our method with a FIFO as default setting for carrying out Link Discovery.

Table 3. Average results of our approach using different caches on LGD (top of the table) and DBP (bottom of the table). F2 stands for FIFO second chance. NA means that the approach timed out (time out = 6 hours)

$\|C\|$	Runtimes (ms)					Hit ratio				
	FIFO	*F2*	*LFU*	*LRU*	*SLRU*	*FIFO*	*F2*	*LFU*	*LRU*	*SLRU*
1,000	4,996.7	7,064.3	8,457.3	14,922.7	17,466.0	0.51	0.50	0.51	0.51	0.51
2,000	5,139.3	7,368.3	9,192.3	15,497.3	17,464.7	0.51	0.51	0.51	0.51	0.50
4,000	5,789.3	7,738.0	9,612.0	15,778.3	18,240.7	0.51	0.51	0.51	0.51	0.51
1,000	8,331.3	11,879.0	NA	8,881.3	12,483.3	0.50	0.50	NA	0.50	0.50
2,000	8,919.0	13,023.7	NA	9,411.0	13,473.3	0.50	0.50	NA	0.50	0.50
4,000	9,866.7	13,684.0	NA	10,385.7	14,431.3	0.50	0.50	NA	0.50	0.50

5.5 Comparison with Existing Approaches

We compared the performance our approach combined with FIFO with that of existing caching approaches, which we used as baselines. To achieve this goal, we measured the overall runtime and hit ratio achieved by our approach with

Table 4. Comparison of GNOME + FIFO with baselines on LGD

Runtimes (ms)						
$\|C\|$	GNOME +FIFO	*FIFO*	*F2*	*LFU*	*LRU*	*SLRU*
1,000	5,974.0	37,161.0	42,090.3	45,906.7	54,194.3	56,904.3
2,000	6,168.0	31,977.0	39,071.3	39,872.0	45,473.0	46,795.0
4,000	7,118.3	21,337.0	40,860.0	28,028.3	26,816.7	27,200.0
Hit ratio						
1,000	0.51	0.17	0.16	0.19	0.17	0.17
2,000	0.51	0.29	0.30	0.32	0.30	0.30
4,000	0.51	0.54	0.55	0.59	0.55	0.56

Table 5. Comparison of GNOME + FIFO with baselines on BDP

Runtimes (ms)						
$\|C\|$	GNOME +FIFO	*FIFO*	*F2*	*LFU*	*LRU*	*SLRU*
1000	9014.3	10276.3	17620	25870.7	40374.7	47713.7
2000	10018.3	11798.7	19588.3	32793	41693	49243
4000	11001.7	13296	21069.7	44744.7	43189	51317.7
Hit ratio						
1000	0.5	0.23	0.23	0.23	0.23	0.23
2000	0.5	0.23	0.23	0.23	0.23	0.23
4000	0.5	0.24	0.24	0.24	0.24	0.24

that of the baselines approaches on the same data as in the precedent section. Our results are shown in Tables 4 and 5. W.r.t. runtime, we are up to an order of magnitude faster. This result is especially significant when one considers the small number of resources in S and T. We achieve a two-fold improvement of the hit ratio, except for high values of $|C|$on LGD, where the hit ratio of our approach is approximately 10 % worse than the other caching approaches but where GNOME is still 3 to 5 times faster than standard caching approaches.

5.6 Scalability

We were also interested in how well our approach scales with a growing amount of data. To measure the approach scalability, we used growing dataset sizes and measured the runtimes achieved by GNOME. The results, presented in Table 6, show our performance with a cache size of 10 % of the dataset size, an optimization time of 2.5 s and a threshold of 0.95. Over all experiments, GNOME achieves a hit ratio of 0.5. Moreover, our results show that the runtime of our approach grows linearly with the number of mappings generated by our approach. For example, on LGD, we generate approx. 360 mappings/ms. Interestingly, this

Table 6. Scalability results of GNOME. k stands for 10^3.

	100k	200k	400k	800k
LGD	362,141.3	1,452,922.0	5,934,038.7	20,001,965.7
DBP	434,630.7	1,790,350.7	6,677,923.0	12,653,403.3

number grows slightly with the size of the dataset and reaches 370 mappings/ms for $|S| = |T| = 8 \times 10^5$ resources.

Overall, the evaluation results presented above allow answering all three of the questions which guided our experimental design: pertaining to Q1, our experiments show clearly that the naïve resp. the best-effort approach are to be preferred over the more complex greedy approaches as they lead to a good tradeoff between runtime and hit ratio. The answer to Q2 is that our approach clearly outperforms all the baseline approaches in all settings. This is a positive result as it means that the overhead generated by clustering the data and determining the right sequence for the clusters pays off. Finally, concerning Q3, the results suggest that GNOME scales linearly with the number of mappings generated and thus scales well.

6 Related Work

This work is a contribution to the research area of LD. Several frameworks have been developed to this goal. The LIMES framework [12], in which GNOME is embedded, provides time-efficient algorithms for atomic measures (e.g., PPJoin+ [22] and $\mathcal{HR}^3$ [11]) and combines them by using set operators and filters. Most other systems rely on blocking. For example, SILK [9] relies on MultiBlock to execute LS efficiently. A similar approach is followed by the KnoFuss system [15]. Other time-efficient systems include [21] 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 [16]. The idea of pre-indexing is also used by ScSLINT [14], who however give up the completeness of results to this end, a concessing we were not willing to make. CODI uses a sampling-based approach to compute anchor alignments to reduce its runtime [8]. Other systems descriptions can be found in the results of the Ontology Alignment Evaluation Initiative [4].[4] The idea of optimizing the runtime of schema matching has also been considered in literature [20]. For example, [17] presents an approach based on rewriting. Still, to the best of our knowledge, GNOME is the first generic approach for link discovery that allows scaling up divide-and-merge algorithms.

Our approach towards improving the scalability of LD is related to clustering, caching, scheduling and heuristics for addressing NP-complete problems. Each of these areas of research has a long history and a correspondingly large body of literature attached to it. Modern graph clustering approaches (see [19] for a survey) will play a central role in our endeavors to parallelize GNOME. The caching

[4] http://ontologymatching.org.

problem was studied in the context of hardware and software development. Surveys such as [1,18] point to the large number of solutions that have seen the day of light over the years. We limited ourselves to caching approaches that are used commonly in this work. Similarly to caching, a large number of approaches have been developed to scale up to the TSP, of which fall into the categories deterministic or non-deterministic [6]. While some rely on using more efficient hardware (see,e.g., [5]), most approaches are heuristics that can be run on any hardware and include approaches ranging from genetic programming (see,e.g., [7]) to collective [3] reinforcement learning [2]. Within this work, we wanted to show that the paradigm underlying GNOME outperforms the state of the art. Hence, we limited ourselves to exploring simple approaches towards addressing the TSP. An inclusion of high-performance TSP solvers remains future work.

7 Conclusion and Future Work

We presented an approach for the efficient computation of large divide-and-merge tasks on commodity hardware. We showed that our approach performs well even on large datasets. Moreover, we showed that the scheduling performed by our approach leads to improved runtimes when compared with state-of-art caching approaches. GNOME can be extended in a number of ways. First, we will consider the parallel processing of this approach. To this end, GNOME will be extended with a partitioning algorithm applied to the task graph. Moreover, we will address the determination of the best possible configuration for our approach (e.g., w.r.t. the optimization time used by the best-effort scheduling algorithm). This will also be the subject of future works.

Acknowledgement. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 688227, the DFG project LinkingLOD and the BMWI project SAKE.

References

1. Ali, W., Shamsuddin, S.M., Ismail, A.S.: A survey of web caching and prefetching. Int. J. Adv. Soft Comput. Appl. **3**(1), 18–44 (2011)
2. Dorigo, M., Gambardella, L.M.: Ant-q: a reinforcement learning approach to the traveling salesman problem. In: Proceedings of ML-1995, Twelfth International Conference on Machine Learning, pp. 252–260 (2014)
3. Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans. Evol. Comput. **1**(1), 53–66 (1997)
4. Euzenat, J., Ferrara, A., Robert, W., van Hage, L., Hollink, C.M., Nikolov, A., Ritze, D., Scharffe, F., Shvaiko, P., Stuckenschmidt, H., Sváb-Zamazal, O., dos Santos, C.T.: Results of the ontology alignment evaluation initiative. In: OM, 2011 (2011)

5. Fujimoto, N., Tsutsui, S.: A highly-parallel TSP solver for a GPU computing platform. In: Dimov, I., Dimova, S., Kolkovska, N. (eds.) NMA 2010. LNCS, vol. 6046, pp. 264–271. Springer, Heidelberg (2011)
6. Goyal, S.: A survey on travelling salesman problem. In: Proceedings of 43rd Midwest Instruction and Computing Symposium (MICS), 2010 (2010)
7. Grefenstette, J., Gopal, R., Rosmaita, B., Van Gucht, D.: Genetic algorithms for the traveling salesman problem. In Proceedings of the first International Conference on Genetic Algorithms and their Applications, pp. 160–168. Lawrence Erlbaum, New Jersey (1985)
8. Huber, J., Sztyler, T., Nößner, J., Meilicke, C.: Codi: combinatorial optimization for data integration: results for OAEI. In: OM, 2011 (2011)
9. Isele, R., Jentzsch, A., Bizer, C.: Efficient multidimensional blocking for link discovery without losing recall. In: WebDB (2011)
10. Nentwig, M., Hartung, M., Ngonga Ngomo, A.C., Rahm, E.: A survey of current Link Discovery frameworks. Semant. Web, 1–18 (2015) (Preprint)
11. Ngonga Ngomo, A.-C.: Link discovery with guaranteed reduction ratio in affine spaces with Minkowski measures. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 378–393. Springer, Heidelberg (2012)
12. Ngomo, A.-C.N.: On link discovery using a hybrid approach. J. Data Semant. **1**, 203–217 (2012)
13. Ngomo, A.-C.N., Kolb, L., Heino, N., Hartung, M., Auer, S., Rahm, E.: When to reach for the cloud: using parallel hardware for link discovery. In: Cimiano, P., Corcho, O., Presutti, V., Hollink, L., Rudolph, S. (eds.) ESWC 2013. LNCS, vol. 7882, pp. 275–289. Springer, Heidelberg (2013)
14. Nguyen, K., Ichise, R.: ScSLINT: time and memory efficient interlinking framework for linked data. In: Proceedings of the 14th Internation Semantic Web Conference Posters and Demonstrations Track (2015)
15. Nikolov, A., D'Aquin, M., Motta, E.: Unsupervised learning of data linking configuration. In: Proceedings of ESWC (2012)
16. Niu, X., Rong, S., Zhang, Y., Wang, H.: Zhishi links results for OAEI. In: OM, 2011 (2011)
17. Peukert, E., Berthold, H., Rahm, E.: Rewrite techniques for performance optimization of schema matching processes. In: EDBT, pp. 453–464 (2010)
18. Podlipnig, S., Böszörmenyi, L.: A survey of web cache replacement strategies. ACM Comput. Surv. (CSUR) **35**(4), 374–398 (2003)
19. Schaeffer, S.E.: Graph clustering. Comput. Sci. Rev. **1**(1), 27–64 (2007)
20. Shvaiko, P., Euzenat, J.: Ontology matching: state of the art and future challenges. IEEE Trans. Knowl. Data Eng. **25**(1), 158–176 (2013)
21. Song, D., Heflin, J.: Automatically generating data linkages using a domain-independent candidate selection approach. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 649–664. Springer, Heidelberg (2011)
22. Xiao, C., Wang, W., Lin, X., Jeffrey, X.: Efficient similarity joins for near duplicate detection. In WWW, pp. 131–140 (2008)