# Using Caching for Local Link Discovery on Large Data Sets

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

AKSW, Department of Computer Science, University of Leipzig, Leipzig, Germany
{mounir,speck,ngonga}@informatik.uni-leipzig.de
http://aksw.org/

**Abstract.** Engineering the Data Web in the Big Data era demands the development of time- and space-efficient solutions for covering the lifecycle of Linked Data. As shown in previous works, using pure in-memory solutions is doomed to failure as the size of datasets grows continuously with time. We present a study of caching solutions for one of the central tasks on the Data Web, i.e., the discovery of links between resources. To this end, we evaluate 6 different caching approaches on real data using different settings. Our results show that while existing caching approaches already allow performing Link Discovery on large datasets from local resources, the achieved cache hits are still poor. Hence, we suggest the need for dedicated solutions to this problem for tackling the upcoming challenges pertaining to the edification of a semantic Web.

**Keywords:** Caching · Link discovery · Semantic web · Linked data

## 1 Introduction

The Web of Data is now an integral part of the Web which contains more than 60 billion facts pertaining to diverse domains including geo-spatial entities, bio-medicine and entertainment.[1] The architectural paradigm underlying the creation of data sources on the Data Web is very similar to that of the document Web and has led to creation of more than 300 knowledge bases, of which the largest pertain to geo-spatial data (LinkedGeoData) and medicine (LinkedTCGA). One of the most demanding steps while publishing data on the Data Web is the creation of links between knowledge bases. Here, the idea is to connect resources across knowledge bases to facilitate the development of applications based on distributed data, e.g., federated query processing and question answering.[2]

Formally, the link discovery problem can defined as follows [10]: Given two knowledge bases $S$ and $T$ as well as a relation $R$, find all the pairs $(s, t)$ such that $R(s, t)$. For example, $S$ could be the set of all cities in DBpedia while $T$ could

[1] http://lod-cloud.net/state/
[2] http://www.w3.org/DesignIssues/LinkedData.html

© Springer International Publishing Switzerland 2015
P. Cimiano et al. (Eds.): ICWE 2015, LNCS 9114, pp. 344–354, 2015.
DOI: 10.1007/978-3-319-19890-3_22

be the set of all provinces in LinkedGeoData while $R$ could be the `locatedIn` relation, which links two resources $s$ and $t$ when the polygon corresponding to $s$ is completely contained in the polygon corresponding to $t$. Computing the set $M \subseteq S \times T$ of pairs that abide by $R$ is quadratic in complexity when addressed in a naive manner. Hence, exisiting framework aim to approximate $M$ by computing the set $M' = \{(s,t) : \delta(s,t) \leq \theta\}$, where $\theta$ is a threshold and $\delta$ is a distance function. Carrying out the computation of $M'$ in a naive fashion is quadratic in time-complexity and linear in space complexity. Hence a large number of time-efficient algorithms for link discovery have been developed over the last years. To the best of our knowledge, all current implementations of such algorithms assume that the data to link (i.e., the sets $S$ and $T$) can be held in memory. Novel works however show that this assumption is erroneous as large data sets such as LinkedGeoData and LinkedTCGA do not fit in the memory of machines used commonly for link discovery.

In this paper, we study how caching algorithms can be used to improve the space behavior of link discovery algorithms and how well the current approaches perform. To this end, we begin by presenting an architecture for combining caching and efficient link discovery approaches based on blocking and filtering. Thereafter, we present the set of caching algorithms. Then, we evaluate the performance of these algorithms on real data using the ORCHID algorithm as link discovery approach. We conclude the paper with a summary of our insights and a discussion of possible future work.

## 2 Caching

In this section, we present how caching can be used for Link Discovery. In particular, we begin by giving a general idea of the use of caching for Link Discovery. Thereafter, we present the caching approaches evaluated in this paper.

### 2.1 Caching for Link Discovery

Most time-efficient approaches for link discovery rely on reducing the number of comparisons of $s$ and $t$ by grouping elements of the source set to $S_i \subseteq S$ and elements of the target set to subsets $T_j \subseteq T$ and only comparing certain $S_i$ with certain $T_j$. For example, two strings have a distance less or equal to 1 w.r.t. to the edit distance if they share at least one letter. If we assume that the resources in $S$ and $T$ are described by their labels and that $\delta$ is the edit distance on labels, then we can group the elements of $S$ by the letters contained in their labels. In this case, $S_i$ would be the subset of $S$ such that the label of each of the resources in $S_i$ contains the i$^{th}$ letter of the alphabet. If we define $T_j$ similarly, then we would not need to compare $S_i$ with $T_j$ if $i \neq j$, leading to several comparisons not having to be carried out at all. The insight behind the use of caching for link discovery is that even when the sets $S$ and $T$ do not fit in memory, single elements of $S$ and $T$ do. Hence, given an element $s$ of $S$, the data necessary to find all $t \in T$ such that $\delta(s,t) \leq \theta$ can be loaded in memory as

required. Elements of $t$ or even whole subsets $T_i$ of $T$ that are commonly used during computations should be cached so as to be read from memory during computations instead of being loaded from the hard drive, which is obviously more time-consuming.

We implemented these insights as follows (see Algorithm 1): Let $A$ be a time-efficient algorithm and $A(s) \subseteq T$ be the set of all elements of $T$ that are to be compared with $s$ according the to algorithm $A$. We iterate over all $s \in S$ and call the function `load`$(A(s))$. This function encapsulates the cache and loads the portions of $A(s)$ that can be found in memory (i.e., in the cache) directly from the memory. The portions that cannot be found in the cache are loaded from external memory (e.g., the hard drive) sequentially and sent to the cache as well as to the `compare` method, which checks each of the loaded $t$ for whether $\sigma(s,t) \geq \theta$ holds.

```
Data: Source S, target T, distance measure δ, distance threshold θ
Result: Set M ⊆ S × T
M = ∅;
for s ∈ S do
    A = load(s);
    for t ∈ A do
        if compare (s,t) == true then
            M = M ∪ {(s,t)}
        end
    end
    return M;
end
```

**Algorithm 1.** Basic caching-based approach to link discovery

### 2.2 Approaches

Caching strategies have several characteristics and can be classified by these [15]. These characteristics are the time since the last reference to an element in the cache (recency), the number of requests to an element in the cache (frequency), the size of an element in the cache (size), the cost to fetch an element (cost), the time since the last modification (modification), the time when an element gets stale and can be evicted from the cache (expiration) [15].

We choose as simple strategies First-In First-Out (FIFO) and First-In First-Out Second Chance (FIFO2ndChance), as a recency-based strategy Least Recently Used (LRU), as a frequency-based strategy Least Frequently Used (LFU), as a recency/frequency-based strategy Segmented Least Recently Used (SLRU) and as a function-based strategy Least Frequently Used with Dynamic Aging (LFUDA).

**FIFO** is a simple strategy. Once the cache is full, the cache element that has been longest in the cache is removed before the insertion of a new element. It is based on the idea of first-in-first-out (FIFO) lists [16].

**FIFO2ndChance** is a modified FIFO strategy in the way that a cache element that have been longest in the cache and was referenced in the past (i.e., used in a previous computation) in the past is removed but inserted again only once so that it gets a second chance. An unreferenced element that have been longest in the cache is removed.

**LRU** is based on the locality of reference and thus tries to predict future accesses to cache elements from previous accesses. The idea is to evict a cache element that led to the oldest hit in the cache. One of the main drawback of this approach is that the cache is not scan-resistant. Still, this is one of the most commonly used approaches [1].

**LFU** is based on a count of the number of accesses to entries in the cache is kept. The cache evicts the entries with the smallest frequency count when necessary. This approach is scan-resistant but does not make use of the locality of reference. The main drawbacks of this approach is that elements that were accessed often in the past and thus having a high count of the number of accesses can remain in the cache even when they are never requested in the future.

**SLRU** extends LRU by splitting the cache into an unprotected (US) and a protected segment (PS), while the former is used for new cache elements the later is reserved for popular elements. Both segments are LRU strategies but only elements in the US are evicted. New elements are inserted into the US and on an access to this element it is moved to the PS. Elements from the PS are moved back to the US as the most recently used element when the PS gets full.

**LFUDA** avoids the cache pollution drawback of LFU with a dynamic aging effect of the cached elements. It calculates a key value $K$ for each element $i$ in the cache with $K_i = F_i + L$ where $F_i$ is the count of the number of accesses of $i$ and $L$ is the running age factor of the cache. $L$ starts at 0 and is updated for each evicted element $e$ to its key value (i.e., $L = K_e$). The strategy evicts the element with the smallest key value from the cache.

## 3 Experiments and Results

The goal behind our experiments was to determine whether current state-of-the art caching algorithms can be used for link discovery. Geographic domain provides large volume of data and require heavy computations. To this end, we assesses the performance of different caching approaches on real data w.r.t. to the runtime they required and the numbers of hits they were able to achieve. In the following, we begin by presenting the experimental setup used for our experiments. Thereafter, we present and discuss our results.

### 3.1 Experimental Setup

In the following, we present the setup used for our experiments.

**Algorithm for Segmentation.** We used the ORCHID [10] algorithm to compute the data segmentations. We used this algorithm for two reasons: First, it is reduction-ratio-optimal and does not tend to overgenerate data segmentations. Moreover, ORCHID can deal with geo-spatial data. This is important because the (to the best of our knowledge) currently largest data set on the Linked Open Data Cloud, i.e., LinkedGeoData, is a geo-spatial data set. Hence, the bias caused by unnecessary comparisons could be minimized while the size of the datasets used in our experiments could be maximized.

**Data Set.** LinkedGeoData dataset was selected because it is the largest data set on the Linked Open Data Cloud. It is a geospatial dataset generated by converting the data from the OpenStreetMap project[3] into RDF. Currently, LinkedGeoData includes approximately 30 billion triples which describe a.o. 3.8 million ways. Within our experiments, we used the differently sized fragments of the dump used in the original ORCHID paper, which contains all CBDs of ways in LinkedGeoData.

**Caching Algorithms.** We used the following approaches during our evaluation: FIFO, FIFO2ndChance, LRU, SLRU, LFU and LFUDA. For all approaches the evict size was set to be one. Variant cache sizes were used including 10, 100, 1K, 10K and 100K.

**Setup.** The evaluation was carried out in two phases. In the first phase, the size of data is $10^4$ resources. Different distance thresholds of 0, 0.1, 0.3 and 0.5 km were used. Increasing the distance threshold results in the up rise of the number of compared polygons. This imposes more polygons to be cached and more computation time. The cache size was assigned to $10^3$ for all caching approaches in the first phase. We selected the best three approaches for the second phase, which was a scalability evaluation. Here, we measured the number of hits and runtime of the approaches. The promoted approaches were opposed to different cache sizes measuring and comparing their run times and revealing the best performed approach. Cache sizes were $10^1$, $10^2$, $10^3$, $10^4$ and $10^5$. In this phase the data size was increased to be $10^5$ resources.

**Hardware.** The evaluation was carried out on a laptop running an Intel CoreTM i7 Quad Core 2.80GHz processor using 8G RAM.

[3] http://www.openstreetmap.org/

### 3.2 Results

In this section we present the results produced based on the aforementioned exponential setup. Figures 1 and 2 show the results of the first series on a sample of LinkedGeoData containing $10^4$ resources, i.e., $10^4$ polygons. The number of cache hits for each of the caching approaches w.r.t. different distance thresholds is presented in figure 1. It is noticeable that LFU achieves the lowest number of cache hits. SLRU also shows lower number of cache hits compared to the rest of the approaches that are almost close in the results. In figure 2, the runtimes of the different caching approach is depicted and it suggests that LRU, SLRU and FIFO have the lowest run times relative to different distance thresholds. It is clear that the lower number of hits the caching approach provides the longer time it takes for fetching or replacing the targeted data. In FIFO fetching and evicting data are performed in time complexity O(1). The simple iteration on polygons indexes implement by ORCHID the number of hits is high. For LRU and SLRU approaches, their implementations tend to avoid time-consuming updates for cache entries. Given that all approaches achieve similar numbers of cache hits the run time turns to be the effective factor in selecting the approaches for the next phase. Tables 1 and 2 give detailed insight on achieved results in this phase. The presented runtime values in Table 1 are in seconds for the sake of readability.

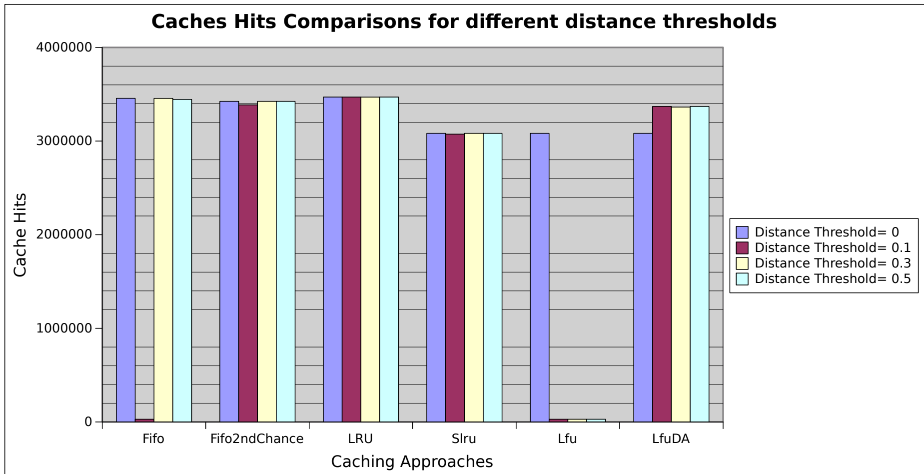


**Fig. 1.** Number of cache hits for different distance thresholds (dataset size = $10^4$ resources)

The results of the second phase for different cache sizes are illustrated in figures 3 and 4. Note that we used a larger dataset of size $10^5$ for this series of experiments. In contrast to the results of previous phase, the runtimes for

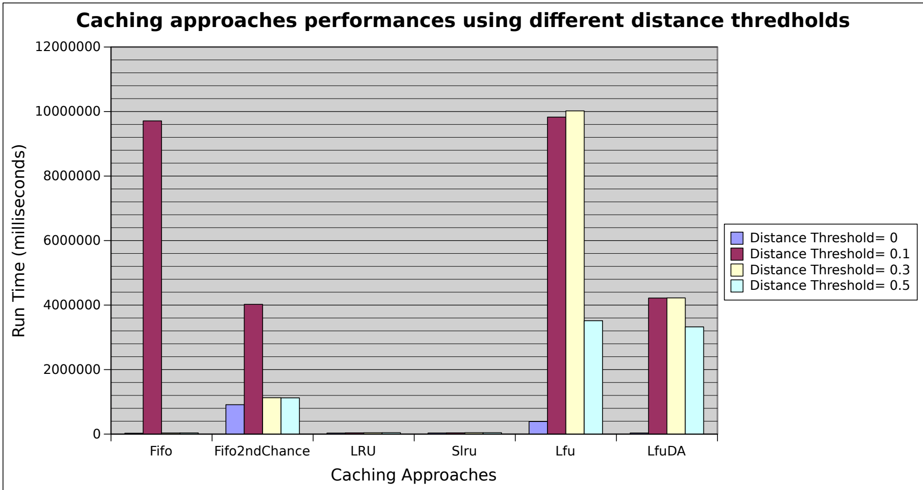


**Fig. 2.** Runtimes of the different caching approaches for different distance thresholds (dataset size = $10^4$ resources)

**Table 1.** Runtimes of the different caching approachesand varying distance thresholds (dataset size = $10^4$ resources)

| CacheType | Distance Threshold= 0 | Distance Threshold= 0.1 | Distance Threshold= 0.3 | Distance Threshold= 0.5 |
|---|---|---|---|---|
| FIFO | 278.4 | 97098.9 | 347.7 | 372.8 |
| FIFO2ndChance | 9120.6 | 40228.7 | 11285.2 | 11242.9 |
| LRU | 321.3 | 386.5 | 423.7 | 440.3 |
| SLRU | 343.8 | 388.1 | 437 | 435.8 |
| Lfu | 3903.2 | 98271.1 | 100202 | 35161.7 |
| LfuDA | 343.8 | 42185.1 | 42216.3 | 33240.6 |

LRU, SLRU and FIFO approaches are quite similar as shown in figure 3. This similarity is due previously mentioned reasons in first phase. Figure 4 shows the superiority of LRU and FIFO in number of hits in accordance with cache sizes. Detailed results are presented in tables 3 and 4.

Overall, FIFO and LRU seem to be test of the caching approaches presented in this paper both in terms of run time and cache hits. However, a careful study of the results they achieve makes clear that their relative hit rates still lie below 50%. This suggests that while current caching approaches do have the potential to reduce the runtime of link discovery approaches, dedicated approaches for link discovery could improve the quality of caching. The study thus suggests that dedicated approaches for link discovery should be investigated in future work to ensure the development of scalable link discovery approaches able to deal with Big Data.

**Table 2.** Number of hits for different caching approaches and varying distance thresholds (dataset size = $10^4$ resources)

| CacheType | Distance Threshold= 0 | Distance Threshold= 0.1 | Distance Threshold= 0.3 | Distance Threshold= 0.5 |
|---|---|---|---|---|
| FIFO | 3456400 | 29052 | 3455933 | 3445072 |
| FIFO2ndChance | 3424230 | 3386093 | 3424230 | 3424270 |
| LRU | 3469912 | 3469912 | 3469404 | 3469871 |
| SLRU | 3082611 | 3073706 | 3082620 | 3082503 |
| Lfu | 3082611 | 29052 | 29059 | 29052 |
| LfuDA | 3082611 | 3369919 | 3363075 | 3369851 |

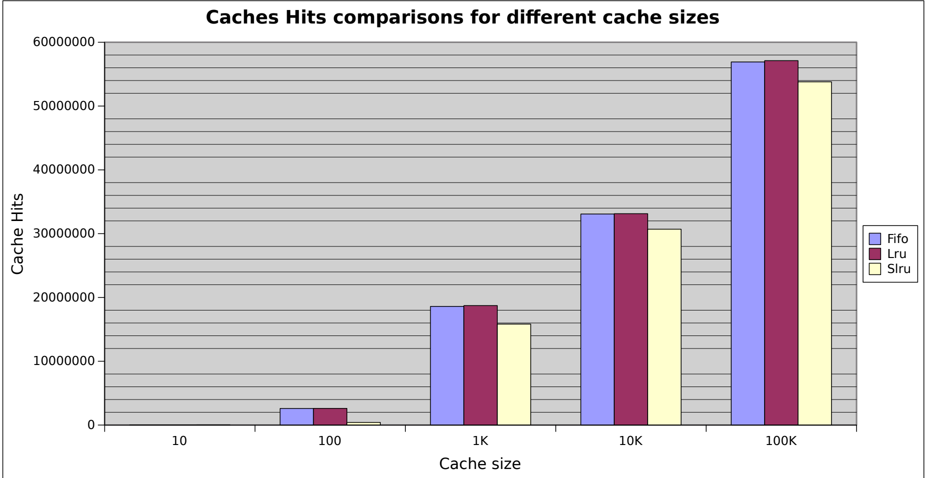


**Fig. 3.** Cache hits for different cache sizes (dataset size = $10^5$ resources)

## 4 Related Work

A vast amount of literature has been produced to elucidate the problem of Link Discovery [7,9,11–13]. Still, with the growth of the size of the dataset at hand, improving the runtime of Link Discovery on large datasets becomes an increasingly urgent problem. Several approaches have been developed with the goal of improving the performance of Link Discovery approaches [8,11–13].

Caching follows the idea to store and reuse as much intermediary knowledge as possible to improve the runtime of the given algorithm. One of the most commonly used approaches is the Least Recently Used algorithm [14]. The idea

**Table 3.** Runtimes (seconds) for different cache sizes(dataset size = $10^5$ resources)

| CacheType | Cache Size= $10^1$ | Cache Size= $10^2$ | Cache Size= $10^3$ | Cache Size= $10^4$ | Cache Size= $10^5$ |
|---|---|---|---|---|---|
| FIFO | 721.7 | 678.8 | 669.9 | 610.8 | 618.4 |
| LRU | 695.7 | 708.9 | 700.8 | 653.6 | 704.3 |
| SLRU | 714 | 907.4 | 658.7 | 694.2 | 700.6 |

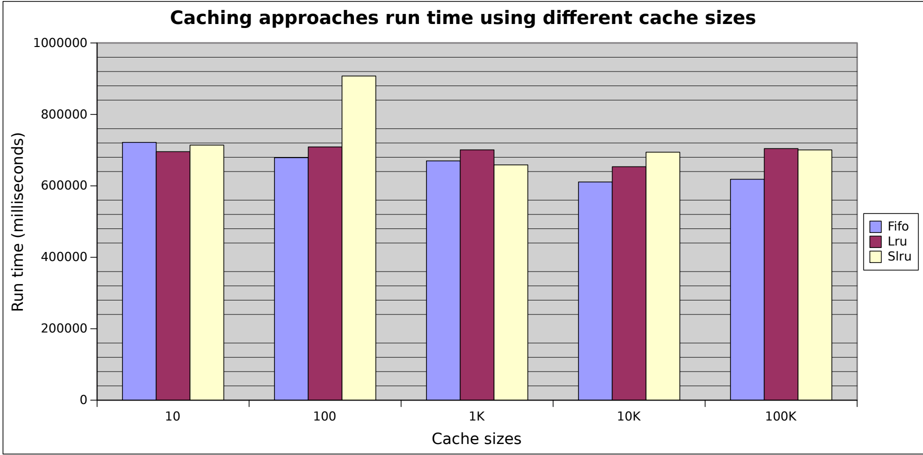


**Fig. 4.** Runtimes for different cache sizes (dataset size = $10^5$ resources)

**Table 4.** Hit rates of different caching approaches for different cache sizes(dataset size = $10^5$ resources)

| CacheType | Cache Size= $10^1$ | Cache Size= $10^2$ | Cache Size= $10^3$ | Cache Size= $10^4$ | Cache Size= $10^5$ |
|---|---|---|---|---|---|
| FIFO | 23927 | 2597652 | 18594739 | 33080982 | 56912799 |
| LRU | 23891 | 2610343 | 18726491 | 33130161 | 57118089 |
| SLRU | 34756 | 412958 | 15822935 | 30696531 | 53798204 |

behind this approach is simply to evict the entry that led to the oldest hit when the cache gets full. One of the main drawbacks of this approach is that the cache is not scan-resistant. Meanwhile, a large number of scan-resistant extensions of this approach have been created. For example, SLRU [6] extends LRU by splitting the cache into a protected and an unprotected area. The Least Frequently Used (LFU) [2] approach relies on a different intuition. Here, a count of the number of accesses to entries in the cache is kept. The cache evicts the entries with the smallest frequency count when necessary. This approach is scan-resistant but does not make use of the locality of reference. Consequently, it was extended by window-based LFU [5], sliding window-based approaches [3] and dynamic aging (LFUDA) [1] amongst others. Another commonly used caching strategy is based on the idea of first-in-first-out (FIFO) lists [16]. When the cache is full, this approach evicts the entry that have been longest in the cache. The main drawback of this approach is that it does not make use of locality. Thus, it was extended in several ways, for example by the "FIFO second chance" approach [16]. Other strategies such as Greedy Dual (GD$^\star$) [4] use a cost model to determine which entries to evict.

## 5 Conclusion and Future Work

In this paper, we presented an evaluation of different caching approaches integrated for link discovery. We used ORCHID as link discovery algorithm and measured the effect of cache sizes, dataset sizes and the distance thresholds on the performance of different caching strategies using real data. FIFO and LRU approaches were determine to be the best approaches for caching data when carrying out link discovery. Still, the relative hit rates of these approaches lie by less than 50%. Hence, our study suggests the need for dedicated caching approaches for link discovery. The development of such approaches will be carried out in future work.

## References


1. Arlitt, M., Cherkasova, L., Dilley, J., Friedrich, R., Jin, T.: Evaluating content management techniques for web proxy caches. SIGMETRICS Performance Evaluation Review **27**(4), 3–11 (2000)
2. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: evidence and implications. In: INFOCOM, pp. 126–134 (1999)
3. Hou, W.-C., Wang, S.: Size-adjusted sliding window LFU - a new web caching scheme. In: Mayr, H.C., Lazanský, J., Quirchmayr, G., Vogel, P. (eds.) DEXA 2001. LNCS, vol. 2113, pp. 567–576. Springer, Heidelberg (2001)
4. Jin, S., Bestavros, A.: Greedydual* web caching algorithm - exploiting the two sources of temporal locality in web request streams. In: 5th International Web Caching and Content Delivery Workshop, pp. 174–183 (2000)
5. Karakostas, G., Serpanos, D.N.: Exploitation of different types of locality for web caches. In: Proceedings of the Seventh International Symposium on Computers and Communications, pp. 207–2012 (2002)
6. Karedla, R., Love, J.S., Wherry, B.G.: Caching strategies to improve disk system performance. Computer **27**, 38–46 (1994)
7. Lyko, K., Höffner, K., Speck, R., Ngomo, A.-C., Lehmann, J.: SAIM – one step closer to zero-configuration link discovery. In: Cimiano, P., Fernández, M., Lopez, V., Schlobach, S., Völker, J. (eds.) ESWC 2013. LNCS, vol. 7955, pp. 167–172. Springer, Heidelberg (2013)
8. Ngonga Ngomo, A.-C.: A time-efficient hybrid approach to link discovery. In: Proceedings of OM@ISWC (2011)
9. 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)
10. Ngonga Ngomo, A.-C.: ORCHID – reduction-ratio-optimal computation of geo-spatial distances for link discovery. In: Alani, H., et al. (eds.) ISWC 2013, Part I. LNCS, vol. 8218, pp. 395–410. Springer, Heidelberg (2013)
11. Ngonga Ngomo, A.-C.: HELIOS – execution optimization for link discovery. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 17–32. Springer, Heidelberg (2014)
12. 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)

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. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The lru-k page replacement algorithm for database disk buffering. SIGMOD Rec. **22**, 297–306 (1993)
15. Podlipnig, S., Böszörmenyi, L.: A survey of web cache replacement strategies. ACM Comput. Surv. **35**(4), 374–398 (2003)
16. Tanenbaum, A.S., Woodhull, A.S.: Operating systems - design and implementation, 3rd edn. Pearson Education (2006)