

How Does the Link Queue Evolve during Traversal-Based Query Processing?

Ruben Eschauzier¹, Ruben Taelman¹ and Ruben Verborgh¹

¹Department of Electronics and Information Systems, Ghent University – imec

Abstract

Link Traversal-based Query Processing (LTQP) is an integrated querying approach that allows the query engine to start with zero knowledge of the data to query and discover data sources on the fly. The query engine starts with some seed documents and dynamically discovers new data sources by dereferencing hyperlinks in previously ingested data. Given the dynamic nature of source discovery, query processing tends to be relatively slow. Optimization techniques exist, such as exploiting existing structural information, but they depend on a deep understanding of the link queue during LTQP. To this end, we investigate the evolution of the types of link sources in the link queue and introduce metrics that describe key link queue characteristics. This paper analyses the link queue to guide future work on LTQP query optimization approaches that exploit structural information within a Solid environment. We find that queries exhibit two different execution patterns, one where the link queue is primarily empty and the other where the link queue fills faster than the engine can process. Our results show that the link queue is not functioning optimally and that our current approach to link discovery is not sufficiently selective.

Keywords

Link Traversal-based Query Processing, SPARQL, Solid, Decentralized Querying

1. Introduction


Link Traversal-based Query Processing (LTQP) [1] is an integrated querying approach where the query engine starts with a set of *seed URIs* and dynamically discovers sources by following hyperlinks discovered in documents of previously dereferenced URIs. The engine internally keeps a queue of discovered URIs (links) it should dereference to obtain documents to query over. This link queue governs the data discovery during LTQP and thus determines in what order we retrieve documents. The order in which we dereference links influences the time it takes to start producing results [2]. If the query engine first dereferences data sources relevant to the query, we can quickly start producing query results, while if we first dereference irrelevant data, the query engine is stuck processing data that will not produce results. The impact of the link queue on LTQP performance makes it an interesting avenue of optimization [2, 3]. However, due to the lack of prior knowledge on the structure of data, the massive size of data accessed [4], and the numerous HTTP requests needed to obtain the data, LTQP is slow [5]. Even with *reachability*

QuWeDa 2023, 7th Workshop on Storing, Querying and Benchmarking Knowledge Graphs at ISWC 2023, Athens, Greece
✉ ruben.eschauzier@ugent.be (R. Eschauzier); ruben.taelman@ugent.be (R. Taelman); ruben.verborgh@ugent.be (R. Verborgh)

🌐 <https://www.rubensworks.net/> (R. Taelman); <https://ruben.verborgh.org/#site> (R. Verborgh)

🆔 0000-0002-6475-806X (R. Eschauzier); 0000-0001-5118-256X (R. Taelman); 0000-0002-8596-222X (R. Verborgh)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

criteria that limit the number of accessed documents and various link prioritization techniques [2], LTQP over the Linked Open Data Web is too slow for many practical applications. The problems of LTQP for Linked Open Data are exacerbated by the significantly faster performance obtained by simply aggregating linked data and serving it as, for example, a SPARQL endpoint. However, for data in decentralized environments with licenses or usage policies, this form of centralization of linked data is impossible, and LTQP becomes an interesting solution. Due to the highly decentralized approach to querying, LTQP is unique because it works for both open and closed-linked data querying. The problem remains that LTQP is slow due to the lack of prior knowledge of the data distribution. Fortunately, some decentralized environments have characteristics we can leverage to improve the status quo. By making structural assumptions on the data based on the characteristics of a decentralized environment, query engines can use prior knowledge for optimization and can guide the order of data discovery. For our analysis, we will use the Solid environment as an example of a decentralized environment of which we have prior knowledge of the data structure.

The Solid protocol describes a linked data publishing paradigm based on access permissions. Depending on the user permissions, some data *should* be inaccessible. The widely used approach to linked data publishing is to harvest data and provide a single access point. However, this is currently impossible for the personal data stored in a Solid pod due to privacy and permission concerns. LTQP can query this data by skipping any document the client can not access.

Furthermore, we know the general topology of the data in Solid. Personal data vaults act as data hubs and predicates known beforehand indicate the structure within data vaults and relationships between data. These predicates serve as link “sources” that contain information on the possible relevancy or priority of the link. To incorporate link source knowledge into query execution, we must understand how the types of link sources are discovered and present in the link queue. To this end, we investigate the diversity of link source types that enter the queue, how many links of each source are in the link queue, and for how long they stay there. This information can guide future LTQP optimization research efforts and uncover unexplored avenues for optimization. Our contributions are as follows:

1. We *analyze the type of links* that enter the link queue and the time of arrival of these links. Thus, providing insight into how LTQP traverses the Solid ecosystem.
2. We *measure the diversity of link types* in the link queue during query execution. Link type diversity indicates the possible merit of link prioritization based on these link types.

While we analyze the Solid decentralized environment, future work can extend our analysis to other decentralized environments.

2. Related Work

In LTQP literature, the focus is on querying the entire Web of linked data without any assumptions on the structural properties of the distributed environment. We will discuss *graph-based*, *intermediate solution-based* [2], and *string similarity-based* algorithms [3].

Graph-based link prioritization [2] applies *vertex scoring* methods to a dynamically constructed directed graph that represents the discovered topology of the Web. Each vertex denotes either a

retrieved document or a queued URI, and directed edges represent that we obtain the URI of the target vertex from the source vertex’s data. The authors use *PageRank* [6] and *Indegree-based scoring* to calculate link priorities. PageRank iteratively scores vertexes according to their importance, where more edges to a vertex will increase its importance, and edges from important vertexes have more weight. Indegree-based scoring counts the number of incoming edges of a vertex and uses this as its importance score.

Intermediate solutions-based link prioritization (ISLP) [2] is an adaptive approach where vertexes are scored based on the number of solutions the vertex has contributed to. ISLP then scores links based on the neighboring vertexes’ scores. This approach uses the belief that highly relevant documents contain links to other relevant documents. *String similarity-based algorithms* [3] also use this assumption. The algorithm prioritizes URIs similar to previously dereferenced URIs that produced query results. To measure URI similarity, the authors use Levenshtein string distance.

For all approaches, the literature suggests that while link prioritization outperforms a simple FiFo queue for some queries, it performs worse for others. We conclude that, while link prioritization has merit, it needs more in-depth research into what makes a sound link prioritization approach and why some queries show no benefit from link prioritization and others do.

3. Experimental Setup

In this section, we first describe the data we use for our analysis. We then introduce the possible link types we can track in the Solid environment and how we measure the link queue evolution. Finally, we introduce metrics to represent the diversity of links in the queue.

3.1. Data

We use the recently introduced *SolidBench benchmark* [7] for our analysis, which uses the *Social Network Benchmark (SNB)* [8, 9] at its core. The SNB models a social network similar to Facebook; social networks produce networks of connected data with users acting as data hubs. This data structure is suited for the Solid protocol, where Solid pods act as data hubs. SolidBench uses this centralized benchmark as a data generator and then fragments this data into data vaults, which function as Solid pods. We use the default data generation and fragmentation parameters, which results in 158,233 RDF files over 1,531 data vaults using the default fragmentation strategy. There are 3,556,159 triples across all files, with 22.47 triples per file on average.

The SolidBench benchmark uses the read-only interactive workload of SNB, which corresponds to the workload that social network applications encounter. The interactive workload of SNB has two query template classes: *complex* and *short*. Preliminary testing [7] of the benchmark shows that these queries are challenging for existing LTQP techniques, with most complex queries reaching timeout. SolidBench includes *discover queries* as a third query template class to enhance SNB. These queries include various choke points in the dataset, like *multi-hop document traversal*, *type index* usage, and determining what HTTP requests are irrelevant. The SolidBench workload consists of 27 query templates, for which we instantiate one query each.

Table 1
Frequency of Special Characters

Non-English or Math	Frequency
http://www.w3.org/ns/ldp#contains	<i>Contains</i>
http://www.w3.org/ns/pim/space#storage	<i>Storage</i>
<i>cMatch</i>	<i>cMatch</i>
http://www.w3.org/2000/01/rdf-schema#seeAlso	<i>SeeAlso</i>
http://www.w3.org/2002/07/owl#sameAs	<i>SameAs</i>
http://xmlns.com/foaf/0.1/isPrimaryTopicOf	<i>TopicOf</i>

3.2. Link Types in the Solid Ecosystem

In the Solid ecosystem, multiple predicates serve a distinct function and result from the structural properties of the ecosystem. For example, *LDP Basic Containers* serve as folders that contain references to documents or other LDP containers, while *type indexes* denote a direct link to a resource in the vault with a type or class, like comments or posts. For a more in-depth explanation of the Solid environment and its structural properties, we refer to [7].

In Table 1, we show all considered predicate types and the short versions of their name. Note the presence of *cMatch*, the reachability criterion described in [4]. We do not consider the *cAll* criterion, as it will lead to an impractical number of followed links. When we dereference a URI, we can record whether we discover this URI from a triple with one of the previously mentioned predicates, thus giving us information on the type of document we are dereferencing.

3.3. Link Queue Analysis

To investigate the behavior of the link queue during LTQP, we will register the link source for each dereferenced link during query execution. We snapshot the link sources in the link queue whenever a link is popped from or pushed to the link queue and register the timestamp. Thus, we obtain the evolution of the sources in the link queue during query execution.

Using this information, we will first plot the different types of sources and their numbers present in the link queue. This plot will give us an understanding of the data discovery pattern of the query engine and allow us to investigate avenues for link prioritization algorithms or other forms of optimization.

To support the visual analysis of the link queue, we introduce metrics that measure link queue characteristics. We are interested in what percentage of all links we obtain from a given source. Furthermore, we will determine the fraction of time the link queue contains k or more links ($pEff(k)$) and calculate the time-weighted average number of link types present in the link queue, given k links are present ($\bar{n}^q(k)$). These metrics give us insight into possible bottlenecks during query execution and whether link prioritization based on link sources has merit for LTQP. We calculate these metrics using

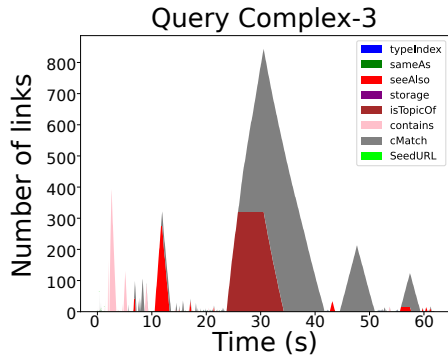
$$pEff(k) = \frac{\sum_{\{t_0 \leq t_i < t_N | n_{t_i}^q \geq k\}} t_{i+1} - t_i}{t_n - t_0}, \quad \bar{n}^q(k) = \frac{\sum_{\{t_0 \leq t_i < t_N | n_{t_i}^q \geq k\}} (t_{i+1} - t_i) n_{t_i}^q}{t_n - t_0}. \quad (1)$$

With t_0 the first timestamp recorded, t_N the final timestamp, and $n_{t_i}^q$ the number of different link sources in the queue at timestamp t_i . These timestamps are obtained whenever a push or pop event happens in the link queue. This implies that T_N signals the end of the query and not a change in the link queue. We execute the queries using Comunica [10]. Comunica uses the algorithm described in [7], which uses a FiFo link queue. Furthermore, we set the query timeout to 60 seconds. The code required to replicate this study is available on Github.

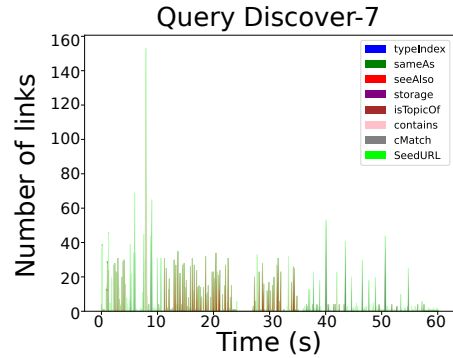
4. Results

In Figure 1, we present a selection of the link queue evolutions that accurately represent what we found in all 27 queries. From these figures, we find two categories of queries: queries where the engine can quickly process the number of discovered links and queries where the number of links followed increases steadily to the point that the query engine cannot handle the number of discovered links. Furthermore, we set the timeout to 1,100 seconds for Figure 1e in order to investigate the spike of *cMatch* links. The result is given in Figure 1f. We find that the *cMatch* criterion can quickly generate a large number of links to follow, slowing down the query execution and making the query infeasible to execute.

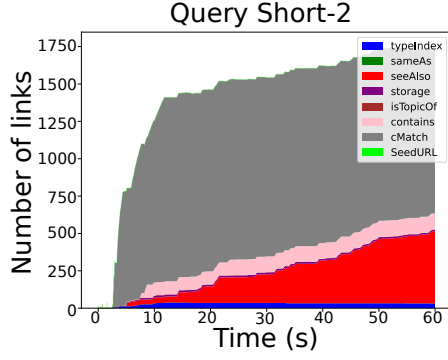
We investigate the metrics introduced in Section 3 to quantify the two query categories. We split the queries into two groups: one with a non-zero number of links in the queue for more than 50% of the query execution time, and the other for less than 50%. The group with high queue occupancy contains twelve queries, while the other group contains fifteen queries. Interestingly, all discover queries belong to the query group with low queue occupancy. The average metrics of the groups are shown in Table 2.



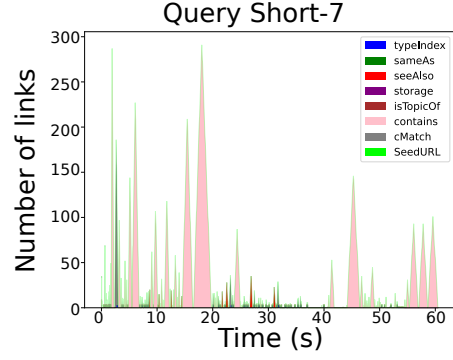
(a) While the query engine manages to fill the link queue, *cMatch* strains the link queue more than others.



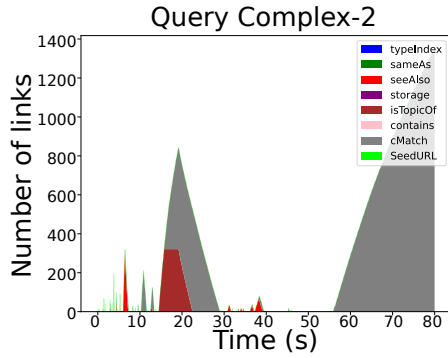
(b) The link queue is primarily empty, but the query times out. The empty link queue shows that current query execution plans are insufficiently optimized.



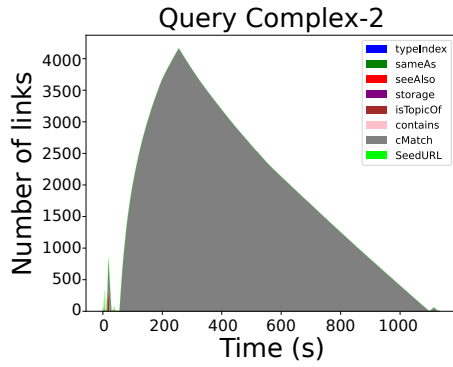
(c) The link queue quickly fills up with many *cMatch* links, and the query engine cannot process the volume of links.



(d) The link queue is primarily empty, but the query times out. The empty link queue shows that current query execution plans are insufficiently optimized.



(e) Before the timeout, we see a spike in *cMatch* links, but the query engine can process the links before this.



(f) When we extend the timeout we find the number of *cMatch* links explodes, making query execution infeasible.

Figure 1: Link queue contents of LTQP engine for five different queries.

Table 2

The average metrics for queries with over and under 50% link queue occupancy. Here $\%cMatch$ denotes the percentage of links with *cMatch* as the source and $\%Contains$ the percentage of links with the *Contains* predicate as the source. $pEff(k)$ denotes the percentage of time the queue has k or more links in it. Finally, $\bar{n}^q(k)$ the average number of links in the queue when at least k links are in the queue.

Query Category	$\%cMatch$	$\%Contains$	$pEff(2)$	$pEff(1)$	$\bar{n}^q(0)$	$\bar{n}^q(1)$
High Queue Occupancy	53.5	26.5	0.447	0.815	2.14	2.445
Low Queue Occupancy	29.0	42.3	0.019	0.118	0.137	1.134

4.1. Discussion

By dividing the queries based on queue occupancy, we find a clear difference in link queue characteristics. Queries with high queue occupancy have a higher percentage of *cMatch* links and, on average, have more than one type of link in the queue for 50% of the query execution time. On the other hand, queries with low queue occupancy have a higher occurrence rate of *contains* links and seldom have more than two types of links in the queue at a given time.

The queries where the link queue is empty for most of the query execution time support the conclusion of previous work that current query plan optimization approaches perform poorly for LTQP [7]. The authors support their argument using an empirical experiment on all discover queries; our results find that these queries all have low queue occupancy. When the link queue is empty, but the query times out, we know that the execution of the query plan over the retrieved data causes the time out and not dereferencing discovered URIs.

For queries with many links in the queue, we find that the link queue often contains different types of links during query execution. This diversity of links indicates that link prioritization strategies based on link sources can influence query execution strategy during LTQP. Furthermore, for queries with many links to follow, the query engine discovers most links using the *cMatch* criterion. The engine primarily uses *cMatch* to traverse to other Solid pods since all data in a single pod can be retrieved using the *contains*, *storage*, and *type* index predicate links [7]. Queries for which the link queue fills up with thousands of *cMatch*-sourced links show that this method of pod discovery is not sufficiently selective. While queries with low queue occupancy support our earlier conclusions [7], our findings for queries with high queue occupancy contradict it. This contradiction is due to the fact we used discovery queries in the empirical experiment to support the conclusion, which all have low queue occupancy.

5. Conclusion

In this paper, we investigated the contents of the link queue during LTQP over Solid pods. Based on the content of the link queue during query execution, we divide queries into two types. The first query type has a primarily empty link queue during query execution. In contrast, the second query type has a link queue that fills rapidly, and the query engine cannot empty it quickly enough. For low queue occupancy queries, the bottleneck is their query execution plan. This bottleneck shows that improving existing zero-knowledge query planning is required to optimize LTQP over the Solid environment. For high queue occupancy queries, we find that while the link queue quickly fills up with links retrieved using *cMatch*, other link sources are often present concurrently.

These observations highlight that *link prioritization* based on the structural properties of Solid can influence query execution time and that the *cMatch* criterion is not sufficiently selective for efficient LTQP. We conclude that future work on LTQP has three avenues for research:

1. Investigate new query optimization algorithms to improve low queue occupancy query execution. Adaptive query planning seems especially promising due to the lack of prior knowledge of the data during LTQP.
2. For high queue occupancy queries, our analysis shows that link prioritization using the structural properties of the Solid system can influence query execution. Thus, future

research into how to use these structural assumptions to prioritize links is an interesting future direction.

3. Finally, the large number of *cMatch* sourced links in the link queue of high queue occupancy queries show, that using *cMatch* to discover new Solid pods is insufficiently selective. Future work should investigate alternative approaches that allow for more selective cross-pod link traversal.

Acknowledgments

This research was supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10). Ruben Taelman is a postdoctoral researcher at the Research Foundation – Flanders (FWO).

References

- [1] O. Hartig, C. Bizer, J.-C. Freytag, Executing sparql queries over the web of linked data, in: The Semantic Web-ISWC 2009: 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings 8, Springer, 2009, pp. 293–309.
- [2] O. Hartig, M. T. Özsu, Walking without a map: Ranking-based traversal for querying linked data, in: The Semantic Web–ISWC 2016: 15th International Semantic Web Conference, Kobe, Japan, October 17–21, 2016, Proceedings, Part I 15, Springer, 2016, pp. 305–324.
- [3] S. J. Lynden, I. Kojima, A. Matono, A. Nakamura, M. Yui, A hybrid approach to linked data query processing with time constraints., LDOW 996 (2013).
- [4] O. Hartig, J.-C. Freytag, Foundations of traversal based query execution over linked data, in: Proceedings of the 23rd ACM conference on Hypertext and social media, 2012, pp. 43–52.
- [5] J. Umbrich, A. Hogan, A. Polleres, S. Decker, Link traversal querying for a diverse web of data, Semantic Web 6 (2015) 585–624.
- [6] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: Bring order to the web, Technical Report, Technical report, stanford University, 1998.
- [7] R. Taelman, R. Verborgh, Link traversal query processing over decentralized environments with structural assumptions, in: Proceedings of the 22nd International Semantic Web Conference, 2023. URL: <https://comunica.github.io/Article-ISWC2023-SolidQuery/>.
- [8] R. Angles, J. B. Antal, A. Averbuch, A. Birler, P. Boncz, M. Búr, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, et al., The ldbs social network benchmark, arXiv preprint arXiv:2001.02299 (2020).
- [9] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, P. Boncz, The ldbs social network benchmark: Interactive workload, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 619–630.
- [10] R. Taelman, J. Van Herwegen, M. Vander Sande, R. Verborgh, Comunica: a modular sparql query engine for the web, in: The Semantic Web–ISWC 2018: 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, Proceedings, Part II 17, Springer, 2018, pp. 239–255.