Personalized Query Engine Optimization for Link Traversal-based Query Processing over Structured Decentralized Environments

Ruben Eschauzier

Ghent University - Imec.

Abstract. The scale of decentralization envisioned for the presently centralized web requires querying approaches that can query numerous small data sources instead of a few large ones. Link Traversal-based Query Processing (LTQP) is a promising candidate for querying highly decentralized environments that executes queries with zero knowledge of the queried data and discovers data sources on the fly. However, as the engine does not know in advance what data will be queried, creating an optimized query plan before executing the query is challenging. Presently, LTQP is employed for client-side querying, where one engine instance services a single client. Despite this, current engines do not utilize client-specific engine query usage patterns to implement personalized query optimization algorithms. This paper will describe the proposed research approach for implementing personalized query optimization techniques, such as caching or learned query optimizers, for LTQP. The objective is to improve query optimization algorithms through the analysis of historical query engine usage, instead of depending on additional prior information. Personalized optimization will be based on existing work in SPARQL optimization literature and fundamental database theory, adapted to LTQP, and aimed at repeating their success in reducing query execution time. As a result, query engines will gain the capability to query large decentralized environments, enabling applications to function within this emerging decentralized web landscape.

1. Introduction

Currently, web user data is stored in centralized silos controlled by massive companies like Facebook, Google, and Amazon. These companies control the data generated by web users, restricting innovation [1]. Rather than adopting a ‘vertical’ approach where all user data from a single service is stored in one central location, various decentralization efforts [2, 3], advocate for a ‘horizontal’ approach. This approach disperses data, resulting in many smaller data entities. For instance, all data pertaining to an individual would be stored in a singular location. As a result, data becomes highly decentralized, spread over possibly millions of sources. Decentralized querying approaches must support cross-source queries, as, for example, social media applications frequently aggregate content from various users to construct a homepage. Initiatives like Solid use the Resource Description Framework (RDF) to store data in a machine-readable format. Traditional SPARQL query algorithms are designed to query a singular RDF store, which is known beforehand. This approach is insufficient

for decentralized environments, as there are multiple sources, and the number of sources is not known beforehand.
Federated SPARQL query algorithms [4, 5] are built to query a few large sources [6, 7] known ahead of time and do not support fine-grained access control. This is insufficient for decentralized environments comprising numerous, small, and highly personal data requiring access control. Instead, querying over these decentralized environments can be done by Link Traversal-based Query Processing (LTQP) [8]. LTQP is an integrated querying approach where the query engine dynamically discovers sources by following hyperlinks discovered in documents of previously dereferenced URIs. This approach allows for fine-grained access control, as an LTQP engine can ignore any document it cannot dereference and continue traversing to new documents. Furthermore, LTQP requires no prior knowledge of the location of data sources, as this is discovered on the fly.
Despite these advantages, LTQP still suffers from significant limitations [9], as discerning relevant from irrelevant data sources during query execution is difficult [10], and query planning without pre-computed statistics often produces suboptimal query plans. Current literature on LTQP considers each query as a separate event without considering the usage of shared engine state between query executions. However, LTQP is a client-side query approach where an engine instance exclusively services a single client. As seen in, for example, browser usage [11], clients exhibit patterns when using applications or browsing the internet. These patterns translate to observable patterns in the queries issued to the query engine. For example, users might primarily use a singular application, which only requires a subset of all data available in the decentralized ecosystem. Additionally, users form sub-communities [12] within applications, which can potentially induce sub-graphs of data that are more frequently accessed by members of the sub-community. Query engines should quantify these patterns and leverage them for significantly improved query optimization. When an engine has already seen a large portion of the data in previous queries, it can use previously computed answers, statistics, and indexes to improve query performance. To address this gap in research, *I will reformulate the query optimization problem from singular queries to a sequence of (possibly) correlated queries.* As such, query engines can apply personalized client-specific query optimizations based on the statistical properties of these patterns to improve average query sequence execution time.

2. State of the Art

First, existing approaches for LTQP optimization must be considered. Then, the following sections will discuss existing SPARQL optimization approaches that can be adapted for personalized LTQP optimization.

2.1. Optimizing LTQP

The literature on LTQP optimization aims to improve the execution plan of queries and the prioritization of query-relevant documents. Identifying query-relevant docu-

ments is relies on link prioritization algorithms, which aim to identify query-relevant documents and access them first. On the other hand, LTQP query planning relies on heuristics [13]. These heuristics, which use no prior knowledge, employ four rules to establish the evaluation order of operators. Firstly, they prioritize triple patterns with a designated seed document, except when the seed document represents vocabulary terms. Moreover, they favor query plans featuring filtering triple patterns in proximity to the seed triple pattern. Finally, they create an order where preceding triple patterns contain at least one query variable of the subsequent pattern.
While multiple algorithms [10] for link prioritization exist for the Open Linked Data Web, none show a definitive advantage over others. However, in structured decentralized environments like Solid, previous work [9], demonstrates improved query execution speed when leveraging structural information inherent to such environments.
These studies usually assume limited prior data knowledge. However, if our engine frequently queries the same dataset, the hypothesis is that leveraging prior knowledge obtained from previous query executions can boost query performance.

2.2. SPARQL Caching Strategies

In SPARQL, server-side caching optimizes query performance by storing and reusing computations [14, 15]. While caching entire query results is possible, most strategies focus on frequently encountered basic graph patterns (BGPs) [15]. These BGPs can substitute joins in query plans and influence join optimization [14]. Canonical labeling algorithms assign distinct labels to isomorphic BGPs, ensuring that all isomorphic BGPs receive equivalent labels [14]. Other server-side caching approaches [15] utilize data summaries to compute join reductions and cache these reductions rather than caching query results. Client-side caching aims to minimize requests to SPARQL endpoints by caching complete query results [16] and implementing proactive query fetching [17]. The efficacy of such strategies heavily depends on the cache hit rate. To decide which queries to prefetch, machine learning techniques predict probable subsequent queries based on the current query [16]. When the cache reaches capacity, cache eviction algorithms, such as Least Recently Used (LRU), remove the least recently requested entry.

2.3. Auxillary Data Structures

In this section, we'll briefly examine data structures used by query engines to optimize query plans. While these are typically precomputed offline, making them impractical for LTQP, caching and dynamically discovering them during LTQP query execution could allow the LTQP engine to use traditional SPARQL optimization strategies. Potential structures include:

- **Dataset summaries**, such as the Vocabulary of Interlinked Datasets (VoID) [18], describe statistical information of the underlying dataset. This information can include the number of triples, distinct subjects or predicates, and the occurrences of predicates.
- **Characteristic sets** [19], which define entities sharing the same predicate set

present in the data. Characteristic sets are instrumental in estimating the cardinality of star-shaped joins, thereby enhancing join planning. These sets can be estimated using sampling techniques [20], reducing the cost of computing them.

- **Approximate Membership Functions (AMFs)**, which determine whether a dataset can potentially contain answers to a query. Examples of AMFs are Prefix-Partitioned Bloom Filters (PPBFs) [21] and the extended Semantically Partitioned Bloom Filters (SPBFs) [22].
- **Indexes**, which are utilized to accelerate the lookup of matching triples to triple patterns. Engines calculate different combinations of SPO indexes depending on their implementation.

2.4. Learned Optimizers

Recent literature on learned query optimizers in relational databases [23, 24] is gaining traction, utilizing reinforcement learning to train the optimizer. Queries are transformed into numeric vectors containing information crucial for query planning, with various featurization methods, such as one-hot encoding join predicates [25] or utilizing advanced graph neural networks on the query graph [23]. The next step involves greedily constructing a join plan to minimize predicted execution cost or latency. To predict latency, ReJOIN [25] employs a feed-forward neural network, while newer approaches use tree-based neural networks to handle the tree structure of join plans [23, 24]. The model is trained to minimize the difference between predicted and actual query latency or cost. While most approaches train optimizers from scratch, Bao [24] augments traditional optimizers by learning to select optimal query hints from a predefined set, reducing training costs significantly while improving over traditional optimizers. In the SPARQL query optimization literature, several cardinality estimation techniques [26] using machine learning are highly successful. Learned optimizers operate under the assumption that there is prior knowledge of the data to be queried, allowing models to be trained offline. However, in the case of an LTQP engine, the data queried is not known beforehand and dependent on the queries issued, rendering offline model training impractical. Thus, the model must learn optimization strategies dynamically as users actively issue queries.

3. Problem Statement and Contributions

Building upon the existing work in Section 2, this thesis will use personalized query optimization to overcome the performance issues outlined in Section 1. Personalized query optimization adapts the optimization procedure to the client-specific query usage patterns of the engine. As such, the engine will keep a state that stores optimization-relevant information. The hypothesis underlying this work is:

- **Hypothesis 1:** Personalized query engines can significantly improve query execution times compared to non-personalized query engines by leveraging client-specific query patterns to improve query optimization.

Before any work on personalized query optimization can proceed, real-world client query usage patterns must be identified. These patterns can include but are not limited to, application data requirements, query requirements for different applications, and data update frequency. After the identification of query patterns, they should be translated to an extensive benchmark that can validate the performance of personalized query optimization algorithms in real-world applications. As such, we define research question I.

- **RQ I:** How do client-specific query patterns manifest in real-world usage scenarios, and how can we accurately capture and represent them within a benchmark?

This thesis will explore two approaches to personalized optimization. The use of caching auxiliary data structures or (intermediate) results, and learned query optimizers. Two natural candidates for cached content are intermediate results sets for queries or auxiliary data structures that improve query planning. However, LTQP uses the *cMatch* criterion [27] to extract links to dereference, changing the queried data depending on what predicates are used in the query. Consequently, queries with overlapping sub-BGPs, but different sets of predicates will dereference different documents during query execution. As reusing the result set for the overlapping sub-BGPs of one query on another can generate wrong results, it is vital to evaluate the effectiveness of caching strategies under these conditions. These conditions lead to research questions II & III.

- **RQ II:** Can (intermediate) result caching be effectively utilized during Link Traversal-based Query Processing to enhance query execution performance when the queried subweb of data changes between queries?
- **RQ III:** How does caching auxiliary data structures during Link Traversal-based Query Processing impact the performance of query execution?

Our caching approaches will build upon and extend methods introduced in Section 2. The primary challenge to overcome in LTQP is the dynamic nature of the queried data, as it can differ between queries, influencing cache validity.
Finally, learned optimizers are a promising candidate for extracting query usage patterns from sequences of queries. Formulating the client query usage patterns as a data-generating process (DGP) and the queries as samples from this process, we can use learning methods to approximate this DGP. However, learning methods can be data and compute expensive and require exploring sub-optimal query plans to learn the entire optimization space. As learned optimizers will be trained in an online scenario where the client actively uses the engine, we must design any solutions with data and compute efficiency in mind. This gives us research questions IV & V:

- **RQ IV** Can training a query optimizer lead to improved query performance in Link Traversal-based Query Processing?
- **RQ V:** Does the query performance benefit of training a query optimizer during query execution outweigh the model training cost?

To answer these research questions, the proposed methods will build upon existing lit-

erature for learned cardinality and join plan estimation. From this foundation, data and compute efficiency approaches will be included to facilitate the usage of learned estimation for LTQP.

4. Research Methodology and Approach

The work in this thesis is divided into three packages. The first aims to answer **RQ I**, the second provides an answer to **RQ II & III**, and finally the third will investigate **RQ IV & V**.

4.1. Identification and Simulation of Client-Specific Query Usage Patterns

Evaluating the effectiveness of personalized query optimization algorithms requires a benchmark simulating real-world query patterns. To identify these patterns, the first step is a literature review on sub-communities in social media networks and the use of linked data in existing applications. This literature review will be used in the creation of a theoretical framework outlining various client query usage patterns in social media and linked data.
An existing benchmark, SolidBench [9], simulates a social network application's data that is fragmented to represent Solid data vaults. Extending SolidBench using the theoretical framework of client query usage patterns allows for simulated client-specific query sequences, representing real-world sub-communities and access patterns. The degree of fragmentation and separation between communities and the probability of within-community queries will be an adjustable parameter to enable the analysis of how varying degrees of client-specific query patterns influence personalized query optimization performance.

4.2. Caching in the Context of Link Traversal-based Query Processing

While caching entire query results is straightforward, caching intermediate results in LTQP is complicated as the queried data changes depending on the query predicates. Intermediate result caches must be aware of the underlying documents that produced these results to allow the engine to identify over what data these intermediate results are valid and what data is not included in the cached results. To answer **RQ II**, an adaptive query planner that can adaptively change its execution plan to include intermediate results that are valid for the currently dereferenced documents is needed. Using cached intermediate results, we can reuse computation, quickly check document cache validity through ETags, and produce first results faster. This query planner will need to consider three cases. The first case is where the intermediate results contain results produced using undiscovered data. In this case, careful pruning of cache elements is required. Second, the cache can contain less data than discovered during query execution. In this case, the query planner should first use all valid intermediate results to quickly produce answers before including the additional data in the query execution to ensure result completeness. Finally, these two cases can occur simultaneously, which will require a combination of the solutions of the previous cases. The

proposed query planner must be adaptive, as the validity of the cache can change as more documents are discovered.

For **RQ III**, an investigation into the effect of the data structures described in Subsection 2.3 is required. Data structures like approximate membership functions, dataset summaries, and characteristic sets can be used to determine whether a document can produce answers to a given query. Documents that will never return results to the query can be pruned, reducing the queried data size and improving query execution times. Additionally, dataset summaries and characteristic sets can be used to improve cardinality estimation as they are discovered. This allows the engine to improve its query plan resulting in reduced computational complexity of the query. Finally, the possible upside of indexes is clear. Currently, these indexes are computed as the engine dereferences a document, which is computationally intensive. If the engine can reuse indexes from previous queries, we bypass the need to re-compute them.

The fundamental risks of all caching approaches are the overhead of maintaining the cache and the possibility of cache invalidation. If searching the cache for relevant entries is too computationally intensive and the cache hit rate is low, the engine will spend more time searching the cache than it saves using cache entries. To account for this risk, this thesis will first investigate caching approaches requiring the least complex cache keys, like document-based caching, or query result caching. After successfully applying the straightforward caching approaches complex tasks will be considered. For cache invalidation, we must account for the possibly rapidly changing data landscape in social media applications. However, even when many *new* posts are added, cached information for old and unchanged content remains valid and can improve query execution performance. Furthermore, even if the query is primarily over the subset of data that rapidly changes, caching the static content in social-media applications can inform the engine of their (ir)relevancy, reducing the queried data size and improving performance. To determine whether a cached entity is valid, we can use the ETag header or introduce data vault server-side data structures that indicate the last change to a resource.

4.3. Learned Query Optimization in Link Traversal-based Query Processing

To answer **RQ IV**, personalized query engines need learned query optimization algorithms that work in an online setting since LTQP engines do not know what data they will query in advance. As such, any offline training algorithm that requires millions of training examples is infeasible. My previous work in SPARQL join order optimization [28] shows that while reinforcement learning-based join order optimization is promising, it is computationally expensive to train an optimizer from scratch. Instead, learned query optimization hint approaches [24, 29] train models that give hints to existing optimizers, like what join operator to use. These approaches require significantly less training time and are thus more suitable for online learning. To answer **RQ IV & V**, relational learned query optimization hints will be adapted for use in LTQP.

5. Evaluation Plan

The evaluation of this work will be done by implementing prototype algorithms on their own, or in combination with other prototypes.

- The prototypes will be built into a modular open-source LTQP query engine [5]. By implementing our approach as modules, other researchers can easily replicate and extend these approaches.
- The primary evaluation method is the benchmark introduced in Subsection 4.1. The benchmark will simulate varying intensities of observed query patterns to determine the impact this has on performance. Further evaluation on different benchmarks will be included if valuable and compatible with LTQP.

The evaluation of query optimization performance will follow evaluation approaches of previous work on LTQP. In practice, the following metrics are often used as gauges of performance:

- **Query execution time**, indicating overall query execution efficiency.
- **First *k* result arrival times**, as LTQP is a streaming querying approach, producing first results quickly improves the client experience.
- **Diefficiency** [30], measures the efficiency of result arrival times during query execution. Engines that quickly produce many results are considered better.
- **Result completeness**, as any caching or document pruning strategy could introduce mistakes, result completeness will be verified and ensured during evaluation.

These metrics will be used to compare the state-of-the-art approaches for non-personalized LTQP optimization to our intended personalized optimization algorithms.
For further analysis of the caching approaches in Subsection 4.2, cache hit rate and overhead will be investigated. The preceding metrics will provide a clear picture of the effectiveness of personalized query optimization for LTQP.

6. Preliminary Results

Initial work was primarily focused on the exploration of the literature around relational database optimization, SPARQL query optimization, caching theory, learned optimization, and more. Following an extensive literature review, the next step was an exploration of the problem space. To facilitate an understanding of LTQP, I produced an early-stage visualization tool of how LTQP engines explore the decentralized environments and an in-depth analysis of the document links discovered during query execution [31]. Moreover, I established a software framework to facilitate the subsequent implementation of LTQP optimization algorithms. This was achieved by enabling the engine to associate metadata with each triple processed during query execution.

7. Conclusion

In this thesis, the query optimization problem is reformulated as an optimization problem over a sequence of correlated queries. These correlations are hypothesized to occur due to client-specific query usage patterns during the usage of LTQP query engines. In this context, personalized query optimization can identify patterns in sequences of queries and use them to adapt the query optimization approach. This thesis aims to use caching and learned query optimizers to identify and leverage patterns in query sequences. As a result, LTQP engines will become more efficient without relying on pre-computed statistics and optimizations. The enhanced efficiency of LTQP engines will subsequently improve the responsiveness and practicality of decentralized applications, thus bringing a decentralized web one step closer.

Acknowledgements. The research for this work has been supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10).

References

1. Verborgh, R.: A Data Ecosystem Fosters Sustainable Innovation (2020).
2. Kuhn, T., Taelman, R., Emonet, V., Antonatos, H., Soiland-Reyes, S., Dumontier, M.: Semantic micro-contributions with decentralized nanopublication services. (2021).
3. Mansour, E., Sambra, A.V., Hawke, S., Zereba, M., Capadisli, S., Ghanem, A., Aboulnaga, A., Berners-Lee, T.: A Demonstration of the Solid platform for Social Web Applications. In: WWW (2016).
4. Saleem, M., Potocki, A., Soru, T., Hartig, O., Ngomo, A.-C.N.: CostFed: Cost-based query optimization for SPARQL endpoint federation. Procedia Computer Science. 137, 163–174 (2018).
5. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a modular SPARQL query engine for the web. In: ISWC. Springer (2018).
6. Dang, M.-H., Aimonier-Davat, J., Molli, P., Hartig, O., Skaf-Molli, H., Le Crom, Y.: FedShop: A Benchmark for Testing the Scalability of SPARQL Federation Engines. In: ISWC. Springer (2023).
7. Qudus, U., Saleem, M., Ngonga Ngomo, A.-C., Lee, Y.-koo: An empirical evaluation of cost-based federated SPARQL query processing engines. Semantic Web. 12, 843–868 (2021).
8. Hartig, O., Bizer, C., Freytag, J.-C.: Executing SPARQL queries over the web of linked data. In: ISWC (2009).
9. Taelman, R., Verborgh, R.: Link traversal query processing over decentralized environments with structural assumptions. In: ISWC (2023).
10. Hartig, O., Özsu, M.T.: Walking without a map: Ranking-based traversal for querying linked data. In: ISWC (2016).
11. Adar, E., Teevan, J., Dumais, S.T.: Large scale analysis of web revisitation patterns. In: SIGCHI conference on Human Factors in Computing Systems (2008).
12. Ferrara, E.: A large-scale community structure analysis in Facebook. EPJ Data

Science. 1, 1–30 (2012).
13. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: ESWC. Springer (2011).
14. Papailiou, N., Tsoumakos, D., Karras, P., Koziris, N.: Graph-aware, workload-adaptive SPARQL query caching. In: SIGMOD (2015).
15. Madkour, A., Aly, A.M., Aref, W.G.: Worq: Workload-driven rdf query processing. In: ISWC (2018).
16. Zhang, W.E., Sheng, Q.Z., Yao, L., Taylor, K., Shemshadi, A., Qin, Y.: A learning-based framework for improving querying on web interfaces of curated knowledge bases. (2018).
17. Zhang, W.E., Sheng, Q.Z., Qin, Y., Yao, L., Shemshadi, A., Taylor, K.: SECF: Improving SPARQL querying performance with proactive fetching and caching. In: Annual ACM Symposium on Applied Computing (2016).
18. Alexander, K., Cyganiak, R., Hausenblas, M., Zhao, J.: Describing linked datasets with the VoID vocabulary. (2011).
19. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: ICDE (2011).
20. Heling, L., Acosta, M.: Estimating characteristic sets for RDF dataset profiles based on sampling. In: ESWC (2020).
21. Aebeloe, C., Montoya, G., Hose, K.: Decentralized indexing over a network of RDF peers. In: ISWC (2019).
22. Aebeloe, C., Montoya, G., Hose, K.: The Lothbrok approach for SPARQL Query Optimization over Decentralized Knowledge Graphs. (2022).
23. Yu, X., Li, G., Chai, C., Tang, N.: Reinforcement learning with tree-lstm for join order selection. In: ICDE (2020).
24. Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., Kraska, T.: Bao: Making learned query optimization practical. In: SIGMOD (2021).
25. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. Presented at the (2018).
26. Schwabe, T., Acosta, M.: Cardinality Estimation over Knowledge Graphs with Embeddings and Graph Neural Networks. (2023).
27. Hartig, O., Freytag, J.-C.: Foundations of traversal based query execution over linked data. In: ACM Hypertext Conferenceand social media. pp. 43–52 (2012).
28. Eschauzier, R., Taelman, R., Morren, M., Verborgh, R.: Reinforcement Learning-Based SPARQL Join Ordering Optimizer. In: ESWC (2023).
29. Woltmann, L., Thiessat, J., Hartmann, C., Habich, D., Lehner, W.: FASTgres: Making Learned Query Optimizer Hinting Effective. (2023).
30. Acosta, M., Vidal, M.-E., Sure-Vetter, Y.: Diefficiency metrics: measuring the continuous efficiency of query processing approaches. In: ISWC (2017).
31. Eschauzier, R., Taelman, R., Verborgh, R.: How Does the Link Queue Evolve during Traversal-Based Query Processing? (2023).