Link Traversal over Decentralised Environments using Restart-Based Query Planning

Jonni Hanski, Simon Van Braeckel, Ruben Taelman, and Ruben Verborgh

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

Abstract. With the emergence of decentralisation initiatives to address various issues around regulatory compliance and barriers of entry to data-driven markets, data access abstraction layers in the form of query engines are needed to assist in developing services on top of such environments. Prior work, however, has demonstrated significant network overhead during data retrieval in traversal-based query execution over decentralised Linked Data sources, dwarfing the relative impact of local processing and query optimisations. Certain decentralisation initatives, however, offer an environment with seemingly sufficient structure to address this, allowing client-side query engines to attain measurable performance improvements through local optimisations. One example is the Solid initiative, offering distributed well-defined user data stores, helping traversal-based query execution approaches in efficiently locating and accessing query-relevant data. Within this work, we demonstrate the impact of client-side adaptive query planning optimisations within structured distributed environments, using the Solid ecosystem as an example, to highlight the potential for tangible improvements in traversal-based execution. Through the implementation of a restart-based query planning technique, we achieve average query execution time *reductions of up to 36%* compared to a baseline of unchanged query plan execution. Conversely, we also demonstrate how such techniques, when applied without robust cost-benefit estimation, can effectively *double* the query execution time. This illustrates the importance and potential of client-side techniques even in such distributed environments, and highlights the importance of further investigation in the direction of these techniques.

1 Introduction

With the emergence of various decentralisation initiatives to address challenges around centralised data storage solutions, ranging from privacy-related legislation to barriers to entry for a given data-driven market, there is also the emerging need for query engine-based abstraction layers to assist developers in creating services on top of such distributed environments, allowing them to write declarative queries to extract the data they need, without having to be aware of the details of its distribution. Thus, the discovery, acquisition and processing of data becomes the reponsibility of the query engine, and for interactive user-facing applications, the engine has to perform these tasks with sufficient user-perceived performance to make such solutions viable in practice [16].

The challenges around data discovery can be addressed through Link Traversal Query Processing (LTQP) [11]. However, even with the varous heuristics of *zero-knowledge query planning* [10], the lack of prior knowledge of the data being queried over may result in suboptimal query plans [18]. And even though the relative impact of the query plan, as compared to the cost of data access over network, may be marginal in some environments [13], in more structured ones it has been shown to be significant [18], with potential for theoretically halving the total execution time. Thus, additional *adaptive query processing* [7] techniques should be explored, for the purposes of adapting the initial plan or its execution to runtime conditions based on feedback, to reach more optimal query plans.

Within this work, we have chosen to employ a client-side restart-based query planning technique over a Solid environment [19], to investigate the impact of adaptive techniques on client-side query processing even in distributed environments with data access overhead, such as with link traversal query execution.

The remainder of this paper is structured as follows. Section 2 briefly discusses related work, followed by Section 3 introducing our research question and hypotheses, as well as Section 4 outlining our approach to tackling them. Section 5 explains our experiments, followed by the results in Section 6. The paper is concluded by our conclusions in Section 7.

2 Related Work

Through widespread adoption of the Linked Data principles, the World Wide Web enables a globally distributed dataspace in the form of *the Web of Linked Data* [12,13]. The *traversal-based query execution* technique [11,10,13], building upon these principles, allows for evaluation of queries over Linked Data using a *follow-your-nose* approach to traversing URIs. This technique allows a query engine to evaluate a SPARQL query over an increasing number of data sources on the Web of Linked Data by intertwining triple pattern matching with link traversal to discover query-relevant data sources on-the-fly [12]. With such a traversal approach, the cost of data retrieval over the network has been shown to marginalise the cost of locally processing that data [13].

Constrained and well-defined data access environments, however, have been shown to increase the relative impact of query planning [18]. The Solid initiative [19] offers one such environment. The initiative seeks to offer individuals greater control over their own data, by storing it in user-controlled permissioned datastores, referred to as *pods*, encouraging and facilitating the reuse of personal data. Notably, pods expose their contents following a set of specifications such as the Solid protocol [3], to assist query engines in data discovery. For this reason, we have chosen Solid as the basis for our experiments, and the SolidBench benchmark for the evaluation to align with existing work.

To take advantage of information discovered during query execution, to overcome limitations imposed by insufficient or inaccurate information on the planning phase, an *adaptive query processing* technique can be used [7]. Such approaches have been categorised as either *inter-query* adaptivity for changes

between executions, or *intra-query* adaptivity for changes during execution. Although inter-query techniques are deemed easier to incorporate into existing *optimise-then-execute* processes, they essentially require executing similar queries over similar data to be able to take advantage of the information acquired [7]. This has been demonstrated through the use of a theoretical oracle in prior work [18], to achieve up to double the query performance of zero-knowledge query planning. Intra-query techniques, on the other hand, aim to take advantage of information as it is discovered *during* the execution of a query plan. Among such approaches are *postponing of plan selection to runtime* [4], as well as various *operator-internal* approaches, for example to allow modifying join operations [6]. Within this work, we demonstrate the potential of intra-query techniques using a restart-based approach detailed in Section 4.

The approaches employed within this work rely on selectivity or cardinality estimates of triple patterns to determine the join plan between them, in an effort to minimise the number of intermediate results. Although centralised storage solutions are often capable of pre-computing such information or providing estimates efficiently, such as through the use of *characteristics sets* [14], within decentralised scenarios this may not always be the case. Thus, various purpose-built estimation techniques have to be applied, such as *variable counting* [15] that estimates the relative selectivities of triple patterns using the type and number of unbound components. Other approaches, such as the set of formulae by Hagedorn et al. [9], make use of the statistics offered in dataset descriptions published using the Vocabulary of Interlinked Datasets (VoID) [5]. Within this work, we have chosen to employ a variable counting approach due to its lack of preconditions, as well as the formulae from Hagedorn et al. due to their suitability for decentralised scenarios with VoID metadata available.

3 Research Question

The purpose of this work is to explore the relative impact of applying client-side adaptive query processing techniques in traversal-based query execution within decentralised environments where data access costs do not dwarf the impact of the query plan, such as with Solid. We use a restart-based approach for this purpose, evaluating the current query plan and restarting it if the plan would differ based on information available during the evaluation. The following research question serves as the basis for our work:

Question 1. Can overall query performance be improved through the application of client-side adaptive techniques, compared to heuristics-based zero-knowledge query planning?

We derived the following hypotheses to answer this research question:

Hypothesis 1. Compared to a heuristic zero-knowledge query planning technique, a restart-based planning approach produces the first and last result faster, and achieves lower total execution time.

Hypothesis 2. Performing plan evaluation and optional restart after uniform amount of execution time for all queries will result in lower performance than baseline for at least a third of the queries.

Hypothesis 3. Performing plan evaluation and optional restart only upon cardinality estimate updates will result in better performance for all queries.

This research question and hypotheses are addressed through a practical implementation and experiments described below.

4 Approach to Client-Side Adaptive Optimisation

We employ an operator-internal technique to restart query plans from the beginning during *pipelined* query execution, where bindings pass through the query plan one by one as they are produced and consumed by the operators, with our wrapper operator encapsulating the query plan by acting as the topmost join in the *tree of joins* of any kind realised by their corresponding *physical operators* that forms the body of the query plan. This wrapper, illustrated in Fig. 1, is responsible for *i)* evaluating at the chosen intervals whether the current query plan is still optimal or not, and *ii)* restarting the encapsulated query plan when the current one no longer appears optimal.

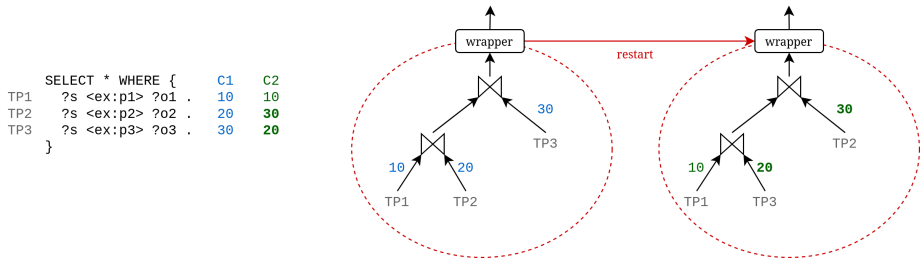

Fig. 1. Once a different plan is produced based on updated cardinalities (TP2, TP3), the wrapper transparently restarts the query plan.

The query plan wrapper operates under bag semantics, under the assumption that a plan, when restarted, produces its full output again from the beginning. The wrapper internally keeps track of all output produced by the plan it encapsulates. Upon restarting the plan, the wrapper uses this record to discard previously produced output, avoiding any spurious duplicates. This record is maintained fully in memory, although practical solutions should consider flushing it to disk as with agjoin [1].

Alongside restarting the query plan, the wrapper operator is also responsible for evaluating the optimality of the chosen plan, by comparing it against a hypothetical plan that would be chosen by the query engine at the time of the

evaluation, given the triple pattern cardinalities available at that moment. If the two plans differ, the wrapper performs the join plan restart, but if they remain identical, then it continues the current execution. The wrapper can be configured to perform its join plan evaluation using two different approaches:

1. *Timeout-based*: When the join plan is initially started, the wrapper sets a timeout. After this timeout, the join plan evaluation is carried out once.
2. *Change-based*: Every time the cardinality estimate of a triple pattern is updated, the join plan evaluation is carried out.

The join plan evaluation relies on cardinality information on triple patterns being updated as new information becomes available. If the cardinality information does not change, the evaluation will produce the same plan every time, and the experiments would be identical. Ideally, any new cardinality estimate would be closer to the true cardinality value known only at the end of the processing. Within this work, we have chosen to employ the following three cardinality estimation techniques:

1. A *variable counting-based approach*, that estimates the relative selectivities of triple patterns and creates a join plan based on these. The initial plan is always created using this approach.
2. A *VoID-based approach*, that uses the VoID dataset descriptions of the Solid pods to estimate the cardinality of that triple pattern using the formulae from Hagedorn et al. [9].
3. A *simplified VoID-based approach*, that functions like approach 2, but assumes the cardinality of a triple pattern to equal the predicate occurrence count.

5 Experiment Setup

The approaches discussed in Section 4 were implemented in Comunica [17], a modular SPARQL query engine framework that provides a baseline link traversal implementation, also previously used to benchmark the relative impact of query plans compared to network overhead in related work [18]. Our implementation is available as open source[1]. Through changes in the query engine configuration, we set up the following test cases to measure the impact of our approaches:

- *Baseline*, as the default configuration of the engine without any of our implementation, doing link traversal with zero-knowledge query planning.
- *Overhead*, with VoID description parsing and cardinality estimation logic in place, but no query plan evaluation or restarts. This measures the overhead introduced by our implementation.
- *Timeout*, identical to the overhead test, but with query plan evaluation taking place once after a set timeout value.
- *Cardinality*, identical to the overhead test, but with query plan evaluation and potential restart taking place every time the cardinality estimate is updated.

[1] https://github.com/surilindur/comunica-components

The dataset and queries were generated using SolidBench[2], a benchmark to simulate a distributed social network use case across Solid pods using the LDBC SNB social network dataset [8]. To support the cardinality estimation, VoID descriptions were generated for each pod.

The experiments were all executed on the same virtual machine, with the server and query engine client running locally, using the *jbr.js*[3] [18] benchmark runner tool. Query timeout was set to 120 seconds following the example set by prior work [18], causing all queries from the complex templates to time out, likewise aligning with such prior work, leaving a total of 75 queries to execute for each configuration. The experiments and our results are available online[4] for reproducibility and validation.

6 Results and Discussion

Beyond query execution time, there was significant variance in the result arrival rates. To capture these differences, we employ the *diefficiency metrics* [2], namely *dief@k*. The *dief@k* value is the integral of result arrivals recorded as a function of time, from the time of 0 results to the time of having produced k results. For our comparison, we have chosen to set k to the *total number of results* for each query, producing the diefficiency value at 100% result completeness. Lower diefficiency values are considered better for the same query with the same results.

From Table 1, one can observe how even the baseline was unable to successfully execute all queries within the allocated timeframe. Thus, we omit these timed-out queries from further analysis. Furthermore, with 63% of queries producing all their results under 5 seconds, and 82% producing them under 20 seconds, the 20-second restart timeout configurations do almost nothing to most of the queries. Additionally, the overhead of the implementation itself appears negligible, and the formulae from Hagedorn et al. perform overall best for cardinality estimation.

The timeout-based approach produced the last result 9–84% slower, but exhibited 64% better diefficiency for queries performing better than baseline. This leads us to accept Hypothesis 2, with a uniform timeout clearly not working for all queries. Evaluating join plans upon cardinality estimate updates produced anywhere between 63% better and 245× worse diefficiency, or 36% lower and 73% higher query execution time. The last result was likewise produced anywhere between 37% faster and 78% slower than the baseline. For the queries performing better than baseline, diefficiency improvements of 40–63% could be observed. Thus, we reject Hypothesis 3 due to update-based restarts not being universally beneficial, but accept Hypothesis 1 due to tangible improvements attainable.

[2] https://github.com/SolidBench/SolidBench.js
[3] https://github.com/rubensworks/jbr.js
[4] https://github.com/surilindur/comunica-experiments

#	Configuration	First result	Last result	Query duration	Below baseline	Above baseline	Average *dief@k*	*dief@k* decrease	*dief@k* increase	Queries finished	Average restarts
1	**baseline**	7.4	9.5	10.5	0%	0%	23.9	0%	0%	**51/75**	0.0
2	**overhead**	8.5	9.9	11.2	43%	54%	22.7	33%	156%	46/75	0.0
3	**overhead simple**	8.3	9.1	10.5	38%	60%	24.5	41%	123%	42/75	0.0
4	**cardinality once**	9.4	10.8	11.9	57%	41%	17.2	47%	3,253%	49/75	0.8
5	**cardinality once void**	**4.6**	6.4	7.3	39%	59%	19.4	40%	1,878%	44/75	0.9
6	**cardinality once void simple**	13.0	17.0	18.2	**93%**	**5%**	37.9	44%	22,744%	43/75	0.8
7	**cardinality unlimited**	8.2	9.3	10.5	71%	27%	18.2	63%	2,660%	48/75	**2.2**
8	**cardinality unlimited void**	**4.6**	**6.0**	**6.8**	50%	48%	**15.2**	52%	808%	42/75	1.3
9	**cardinality unlimited void simple**	7.7	9.7	11.1	**93%**	**5%**	22.9	52%	24,555%	41/75	1.3
10	**timeout 100**	6.8	10.7	13.4	89%	11%	110.2	37%	**189%**	36/75	1.1
11	**timeout 100 void**	5.6	10.5	13.8	72%	28%	173.6	40%	196%	32/75	1.0
12	**timeout 100 void simple**	9.8	17.3	21.1	35%	65%	183.4	49%	403%	31/75	0.9
13	**timeout 1,000**	9.7	14.5	17.6	17%	83%	144.1	**64%**	231%	29/75	0.7
14	**timeout 1,000 void**	5.9	10.6	13.2	61%	39%	138.4	47%	212%	33/75	0.9
15	**timeout 1,000 void simple**	10.5	17.5	20.6	71%	29%	163.9	62%	503%	34/75	0.8
16	**timeout 2,000**	9.3	13.4	15.8	71%	26%	77.2	49%	190%	34/75	0.6
17	**timeout 2,000 void**	7.4	13.9	16.3	22%	75%	112.3	48%	392%	36/75	0.8
18	**timeout 2,000 void simple**	8.1	15.1	18.3	26%	74%	167.6	50%	326%	35/75	0.7
19	**timeout 5,000**	10.6	16.2	18.5	17%	80%	140.3	50%	465%	35/75	0.5
20	**timeout 5,000 void**	8.4	14.3	17.1	73%	24%	140.1	44%	317%	37/75	0.6
21	**timeout 5,000 void simple**	8.5	15.4	17.9	65%	35%	181.9	53%	347%	37/75	0.4
22	**timeout 10,000**	8.7	12.7	14.9	18%	77%	119.1	40%	9,850%	44/75	0.4
23	**timeout 10,000 void**	9.4	13.5	17.5	18%	78%	91.7	40%	3,442%	45/75	0.4
24	**timeout 10,000 void simple**	9.7	15.1	17.8	22%	78%	149.2	41%	325%	37/75	0.4
25	**timeout 20,000**	7.0	10.4	14.6	69%	25%	94.3	24%	12,381%	**51/75**	0.2
26	**timeout 20,000 void**	9.1	13.4	17.8	0%	0%	124.1	37%	13,497%	49/75	0.3
27	**timeout 20,000 void simple**	10.3	16.2	20.0	24%	74%	141.8	32%	386%	38/75	0.3

Table 1. Overview of the benchmark results for different configurations: average time taken to produce the first and last result (s), average total query duration (s), average *dief@k* when k is the total number of results, the share of finished queries for which *dief@k* was below or above the baseline, the average decrease or increase of that metric relative to baseline for those queries respectively, as well as the he number of queries finished successfully and the average number of join restarts per query execution.

7 Conclusions

In related work [18], the *theoretical* impact of better query planning was shown. In our work, we proved this theory using a restart-based query planning approach, and achieved average reductions of up to 36% in query execution time. Our results show that client-side approaches are instrumental in achieving the levels of query performance needed for real-world interactive applications over decentralised environments such as Solid. This brings us a step closer towards addressing the challenges around privacy and data management at scale, and to lower the barriers of entry to the data-driven market.

Acknowledgements. The described research activities were supported by SolidLab Vlaanderen (Flemish Government, EWI and RRF project VV023/10). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N).

References

1. Acosta, M., Vidal, M.E., Lampo, T., Castillo, J., Ruckhaus, E.: ANAPSID: an adaptive query processing engine for sparql endpoints. In: International Semantic Web Conference. pp. 18–34 (2011)

2. Acosta, M., Vidal, M.E., Sure-Vetter, Y.: Diefficiency metrics: measuring the continuous efficiency of query processing approaches. In: International Semantic Web Conference. pp. 3–19 (2017)
3. Capadisli, S., Berners-Lee, T., Verborgh, R., Kjernsmo, K.: Solid protocol 0.10.0. W3C community group technical report, W3C (2022), https://solidproject.org/TR/2022/protocol-20221231
4. Cole, R.L., Graefe, G.: Optimization of dynamic query evaluation plans. In: ACM SIGMOD international conference on Management of data. pp. 150–160 (1994)
5. Cyganiak, R., Alexander, K., Zhao, J., Hausenblas, M.: Describing linked datasets with the VoID vocabulary. W3C note, W3C (2011), https://www.w3.org/TR/2011/NOTE-void-20110303/
6. Deshpande, A., Hellerstein, J.M.: Lifting the burden of history from adaptive query processing. In: International conference on Very large databases. pp. 948–959 (2004)
7. Deshpande, A., Ives, Z., Raman, V., et al.: Adaptive query processing. Foundations and Trends in Databases **1**(1), 1–140 (2007)
8. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The ldbc social network benchmark: Interactive workload. In: PACM SIGMOD International Conference on Management of Data. pp. 619–630 (2015)
9. Hagedorn, S., Hose, K., Sattler, K.U., Umbrich, J.: Resource planning for sparql query execution on data sharing platforms. In: 5th International Conference on Consuming Linked Data. pp. 49–60 (2014)
10. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: Extended Semantic Web Conference. pp. 154–169 (2011)
11. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL queries over the web of linked data. In: International Semantic Web Conference. pp. 293–309 (2009)
12. Hartig, O., Langegger, A.: A database perspective on consuming linked data on the web. Datenbank-Spektrum **10**, 57–66 (2010)
13. Hartig, O., Özsu, M.T.: Walking without a map: optimizing response times of traversal-based linked data queries (extended version). arXiv preprint arXiv:1607.01046 (2016)
14. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In: 2011 IEEE 27th International Conference on Data Engineering. pp. 984–994 (2011)
15. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL basic graph pattern optimization using selectivity estimation. In: 17th international conference on World Wide Web. pp. 595–604 (2008)
16. Taelman, R.: Towards applications on the decentralized web using hypermedia-driven query engines. ACM SIGWEB Newsletter (Oct 2024)
17. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: A Modular SPARQL Query Engine for the Web. In: International Semantic Web Conference. pp. 239–255 (2018)
18. Taelman, R., Verborgh, R.: Link traversal query processing over decentralized environments with structural assumptions. In: International Semantic Web Conference. pp. 3–22 (2023)
19. Verborgh, R.: Re-decentralizing the web, for good this time. In: Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web, pp. 215–230 (2023)