FACULTY OF SCIENCES

Using Discovery of Pre-Generated Metadata with Adaptive Query Planning to Improve Query Performance over Decentralized Environments

Simon Van Braeckel
Student number: 01702644

Supervisors: Dr. ir. Ruben Taelman, Prof. dr. Pieter Colpaert
Counsellors: Prof. dr. ir. Ruben Verborgh, Ir. Wout Slabbinck, Arthur Vercruysse, Jonni Hanski, Bryan-Elliott Tam

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2022-2023

GHENT UNIVERSITY

Acknowledgments

I am deeply grateful to Ruben and Jonni for their guidance, support, and consistent advice which has been invaluable throughout this journey. Their willingness to assist whenever I faced challenges was a true testament to their dedication. They not only helped me move forward during difficult times but also fostered an environment of collaborative learning that greatly enriched my experience. It is their exceptional mentorship that made thesis possible.

I extend my heartfelt thanks to IDLab for providing me with access to their outstanding facilities. Being surrounded by enthusiastic and like-minded individuals in such an inspiring environment played a significant role in keeping me motivated and focused during my thesis work.

Lastly, I want to express my deepest appreciation to my friends and family for their unwavering support and patience throughout this endeavor.

Together, the guidance of Ruben and Jonni, the support of IDLab, and the love and encouragement of my friends and family have been instrumental in shaping this thesis. I am truly grateful to everyone who has played a part in this journey, and I could not have accomplished this without each and every one of you.

Usage

Summary

Current decentralization efforts such as Solid offer a higher degree of decentralization than ever before, a development with which querying techniques have struggled to keep pace.

As integrated querying techniques are outperformed twofold by their two-phase counterparts, the lack of upfront information available in these environments begins to show it's issues. To bridge this gap, there is a growing need for better query planning in this context.

We developed extensions to the integrated Link-Traversal-based approach, facilitating the gathering of planning-relevant information, particularly cardinality estimates, and updating the query plan during execution.

We demonstrate an average performance improvement of 32.57% as a result of providing up front information to the query engine, showing that simple methods of providing up-front information provide a valuable basis for future work on query planning in decentralized environments.

Samenvatting

Huidige decentralisatie-initiatieven zoals Solid bieden een hogere mate van decentralisatie dan ooit tevoren, een ontwikkeling waarop querytechnieken achterop lopen.

Het gebrek aan vooraf beschikbare informatie voor query planners in deze omgevingen begint zijn problemen te tonen, aangezien geïntegreerde querytechnieken twee keer slechter presteren dan hun tegenhangers. Om deze kloof te overbruggen is er een groeiende behoefte aan betere query-planning in deze context.

We hebben uitbreidingen ontwikkeld voor de geïntegreerde Link-Traversal-gebaseerde aanpak om het verzamelen van planningsrelevante informatie, met name kardinaliteitsschattingen, en het bijwerken van het queryplan tijdens de uitvoering mogelijk te maken.

Door deze informatie vooraf te verstrekken aan de query engine bekomen we een gemiddelde prestatieverbetering van 32,57%, waaruit blijkt dat eenvoudige methoden voor het vooraf verstrekken van informatie een waardevolle basis vormen voor toekomstig werk op het gebied van queryplanning in gedecentraliseerde omgevingen.

Using Discovery of Pre-Generated Metadata with Adaptive Query Planning to Improve Query Performance over Decentralized Environments

Simon Van Braeckel
Supervisors: Dr. ir. Ruben Taelman, Jonni Hanski, Prof. dr. Pieter Colpaert, Prof. dr. ir. Ruben Verborgh

***Abstract*—Current decentralization efforts such as Solid offer a higher degree of decentralization than ever before, a development with which querying techniques have struggled to keep pace. As integrated querying is significantly outperformed by its two-phase counterpart, the lack of upfront information available in these environments begins to show it's issues. To bridge this gap, there is a growing need for better query planning in this context. We developed extensions to the integrated Link-Traversal-based approach, facilitating the gathering of planning-relevant information, particularly cardinality estimates, and updating the query plan during execution. We demonstrate an average performance improvement of 32.57% as a result of providing up front information to the query engine, showing that simple methods of providing up-front information provide a valuable basis for future work on query planning in decentralized environments.**

***Keywords*—Adaptive, Query Planning, Semantic Web, Linked Data, Link-Traversal, Join ordering**

I. Introduction

The web has become increasingly centralized in recent years. To counter this development, decentralization initiatives like Solid [4], which attempt to put data control back in the hands of end-users, have gained popularity. Currently, there is a lack of efficient querying techniques for such heavily decentralized contexts, due to most existing techniques focusing on the use case of centralized environments [1]

Link-Traversal-based Query Processing (LTQP) [2] is an approach that shows promise in this context, but has not been widely adopted in practice due to performance concerns related to lack of upfront information about the data. This thesis aims to address these performance limitations by extending the existing zero-knowledge query planning technique [3]. Our approach involves gathering planning-relevant information, such as cardinality estimates, and updating the query plan accordingly during execution.

In the next section, the relevant related work will be discussed, after which our complete solution is presented. Next, we explain the use case on which the experiments are based. The results are presented thereafter. Finally, some concluding remarks will be made, followed by suggestions for future work.

II. Related Work

By examining the current state of research, we can identify gaps and opportunities for further advancements.

A. Link-Traversal Query Processing

Link-Traversal-based Query Processing (LTQP) [2] is a querying paradigm that was introduced as a way to query the Web of Linked Data, without the need to first index it in a single location [1]. In LTQP, the sources are not known in advance. Instead, data source discovery and query execution happen simultaneously.

Compared to traditional query processing, integrated approaches like LTQP cannot rely on pre-execution optimization algorithms that require prior dataset statistics. Instead, it uses a zero-knowledge query planning technique [3] to order triple patterns based on link traversal-specific heuristics.

According to related work [1], the main area of improvement within LTQP is related to the query plan rather than the discovery of relevant documents.

1) Evaluation of Link-Traversal Query Execution over Decentralized Environments with Structural Assumptions: This related paper presents a comparison between an integrated approach and a two-phase approach in query execution. The two-phase approach acts as a theoretical case, it makes use of an oracle which provides all query-relevant links, allowing it

to leverage traditional optimize-then-execute [6], [7] query planning. The two-phase approach is on average two times faster compared to the integrated approach, which can be attributed to its ability to perform traditional query planning, leveraging cardinality estimates among other information. This potential for significant performance improvements highlights the need for more effective query planning strategies. Our work aims to bridge the gap by changing the query plan during execution.

B. Comunica

Comunica [9] is an adaptive query engine framework which facilitates putting together a query engine that performs integrated query planning and execution. The implementations contributed in this thesis have been created as extensions upon Comunica.

III. Related Concepts

A. The Query Process

Query processing traditionally involves three key steps: parsing, planning, and execution. The first step, parsing, is where the SPARQL query string is transformed into an algebraic expression. The second phase, the planning phase, transforms the algebraic expression into a query plan. A query plan is a step-by-step manual that specifies the sequence of operations required to retrieve the desired data. After obtaining the query plan from the planning phase, it is executed in the third and final phase of the query process.

The separation of query planning and execution leads to suboptimal query execution times in heavily decentralized environments, where optimization of the query plan is hindered since it relies on up-front information about the data.

As mentioned in the introduction, LTQP is an important query processing technique within these environments. It copes with the fact that relevant data sources are unknown before query execution by integrating source discovery and execution. However, the naivety within LTQP has been shown to lead to suboptimal query plans, making it twice as slow as a solution where the query plan is optimal [1].

To bridge this gap, in this thesis we describe using adaptive query processing techniques, which leverage runtime feedback and adjusts the plan or scheduling space [5] during execution, thus tightly intertwining the query planning and execution phases. Adaptive techniques are promising in this case because they facilitate the use of better query plans. By changing the query plan as more information about the data becomes available during execution, the query plan may become closer to the ideal.

We introduce adaptivity through the re-ordering of join operation entries. A join operation refers to the process of combining triple patterns within a query. Triple patterns are combinations of values or variables for subject, object, and predicate, imposing a specific structure on triples. These triple patterns often contain common variables, requiring them to be joined, where bindings for a variable of one triple pattern are matched with bindings for the same variable in another triple pattern. We choose the correct order of join entries using their cardinality, which is the amount of bindings present in the data.

By carefully selecting the order in which the entries are joined, we can reduce the generation of intermediate results and avoid unnecessary work, which leads to faster query execution.

Unfortunately, as the exact cardinalities of join entries are not known in in our use case, we work with estimate cardinalities. To use these estimated cardinalities to order join entries, we assume that it's best to join the entries with the smallest cardinalities first. These are likely to lead to a small number of results, reducing the amount of bindings in subsequent joins, in turn reducing the amount of work required in total.

As join operations occur on triple patterns, knowing the exact cardinality of a triple pattern enables the query engine to make accurate predictions regarding intermediate results during join operations. As a result, TP cardinalities are an ideal criterion for sorting join entries and optimizing query performance.

IV. Use Case

We have tested out solutions with the Solidbench [1] benchmark, which builds upon the well-established Social Network Benchmark (SNB) [10], [11]. It models a query workload and data structure as one would find in a social network scenario, fragmented to simulate decentralized workloads similar to those found in Solid. While this benchmark simulates the context of Solid, the solutions are generalizable to other Linked Data environments.

V. Solution

In this section, we present three distinct configurations that aim to improve query execution in decentralized environments.

A. Counting Triple Pattern Cardinalities & Timeout

While cardinalities of join entries are important for making informed join order decisions, such information about the data is unavailable in a distributed, decentralized context. To address this limitation, we implemented a method of counting the occurrences of

triple patterns (TP) as they appear while also introducing a timeout mechanism. This timeout mechanism restarts the join operation after a certain amount of time, providing the potential for more accurate TP cardinality information to be available.

The main advantage is this approach's independence of the existence of any pre-generated data. This independence enhances the flexibility and adaptability of the query engine.

The second advantage of this approach that it allows the usage of TP cardinalities without demanding the use of excessive disk space for generating them and saving them in a file prior to query execution.

B. Predicate Cardinality File

While TP cardinalities are valuable, creating a file containing the cardinality of each possible Triple Pattern is infeasible in many real-world scenario's due to the sheer volume of possible combinations in a Solid pod. However, it may be possible for smaller Solid pod's, and using the correct compression algorithms. Discussed in our future work section. We turn to predicate cardinalities as a middle ground, using the cardinality of each join entry's predicate as an approximation of the TP cardinality. We investigate whether this approximation is sufficiently accurate to result in performance improvements.

This configuration obtains predicate cardinalities at the beginning of query execution. After the cardinalities have been retrieved, subsequent execution continues as usual, and the newly discovered cardinalities are used to sort future join entries. We investigate whether this approximation is sufficiently accurate to result in performance improvements, and assess the impact of having access to the cardinalities right from the outset.

C. Combining Predicate Cardinality File and Timeout

The third approach combines predicate cardinalities with triple pattern cardinalities. We start our execution by fetching predicate cardinalities from the Solid pod. During query execution, as Triple Pattern cardinalities are counted, we use them when predicate cardinalities are not available for a given join entry.

VI. Experimental Design

We evaluated six distinct approaches, including the baseline approach and a variation of it. The evaluated approaches are as follows: nosep

- **base_zero**: Zero-knowledge based query planning. [3] Also referred to as baseline approach in the remainder of this document.
- **base_card**: Baseline approach, changed to employ a cardinality-based sort algorithm. This approach was compared to the zero-knowledge query planning approach to assess its effectiveness.
- **index_card**: Extension to the baseline, making use of predicate cardinalities as detailed in Section V-B.
- **index_zero**: Index_card approach, instead equipped with the zero-knowledge sort. Used to verify the effectiveness of sorting by predicate cardinalities.
- **count**: Extension to the baseline which counts Triple Patterns and restarts joins, as detailed in Section V-A.
- **count+index**: count approach combined with index_card approach, as detailed in Section V-C.

We employed the SolidBench [1] benchmark described in Chapter IV. However, we found that each of the complex queries resulted in a timeout. As a result, we omitted these complex queries from our analysis. We executed each remaining query three times and calculated average metrics to obtain reliable results with reduced noise. To prevent excessively long query execution, we set a timeout of 1 minute.

Our experiments were conducted on a 64-bit Pop!_OS 22.04 LTS machine equipped with an 8-core Intel i7-10510U 1.80 GHz CPU and 16 GB of RAM. Both the server serving the Solid data and the client performing the queries were executed on the same machine.

We measured execution times for each query and configuration, the full results can be found in the full thesis. The next section shows our summarized results.

VII. Results

A. Subsetting the query set

We identified certain query categories that warranted exclusion from further analysis based on their characteristics.

• Queries *discover-3*, *discover-4* and *short-4* exhibited exceptionally low execution times in the baseline configuration. As a result, variances which might not hold much significance can result in a substantial percentage change in execution time. Including these queries in our calculations severely skewed the averages.

• The baseline approach experienced timeouts in queries *short-2*, *short-3*, and *short-7*, which complicates comparison with these approaches.

• For queries *short-1*, *short-3* and *short-5* our "count" and "count+index" approaches failed to retrieve the correct number of results.

• Query *discover-8* exhibited higher execution times with the index approach. The cause is that this query accesses multiple Solid pods during its execution. This

is something our implementation was not specifically tailored to. As such, the predicate cardinalities from the starting pod would still be used to guide join order decisions, while the data are coming from the new pod. We exclude this query from future evaluations and mention the important use case it represents as our main point for future work.

B. Discussion

1) There is no globally optimal timeout value: In our counting-based approach, we utilize a timeout value to restart join operations after a certain amount of time has elapsed. Figure 1 shows how the execution time of the "count" method evolves depending on this timeout value.

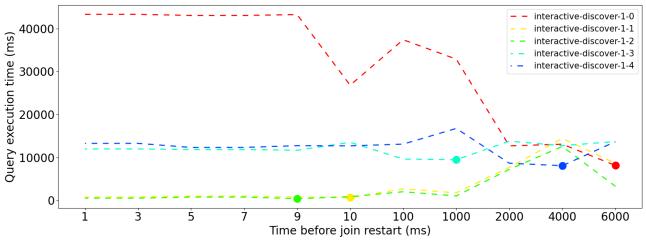


Figure 1: Optimal timeout value and execution time for query discover 1 in configuration timeout without index

This outcome emphasizes the absence of a universally optimal timeout value for all queries. In fact, employing a singular timeout value across all join operations within a single query might be unrealistic, given the distinct nature of each join operation. Instead, we evaluate the execution time of this configuration using the timeout that nets its best execution time. In our future work section, we discuss how this approach can be improved upon.

2) Interference in timeout measurements: A substantial number of queries exhibit ideal timeout values that surpass the execution time. In these cases, there is no occurrence of a join being restarted. We can conclude that these particular cases reap no benefit from join restarts, which implies that the counted cardinalities adversely affect the join order, or the overhead of restarting is excessive. However, this conclusion might be considerably influenced by the uniform application of the same timeout value to all joins. For instance, a query possessing a join that would ideally benefit from a 10ms timeout might exhibit better execution times when another timeout value is employed, solely because the other join operations do not gain from restarting at the 10ms interval. This facet requires more thorough investigation.

3) Analysis using the representative subset: This table compares different approaches against the baseline. The first two columns display the percentage of queries for which each approach exhibits execution times at least 10% higher or lower than the baseline, while the last column displays the average improvement in execution time as a fraction of the baseline approach's performance.

	% of queries		
	better	worse	average improvement
index_zero	0.00%	5.33%	-6.79%
index_card	20.00%	1.33%	32.57%
count	5.33%	5.33%	0.23%
count+index	14.67%	1.33%	29.39%

Table I: Percentage of queries from representative subset with execution time of at least 10% better (resp. 10 worse) than the baseline

Figure 2 shows the amount of queries for which each approach had the best execution time.

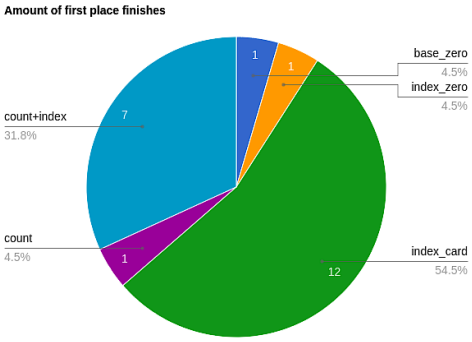


Figure 2: Amount of queries where configurations are the fastest

The results using the representative subset (Section VII-A) demonstrate that the "index_zero" approach is less effective compared to the baseline approach, resulting in an average performance decrease of 6.79%. The evidence supports the importance of sorting join entries with cardinality information for achieving better query plans. Future work may determine whether the approach's negative impact on performance is attributed to the overhead in our implementation of index discovery.

The "index_card" approach stands out as the most effective among the evaluated techniques. On average, it outperforms the baseline "index_zero" approach by a substantial 32.57%. The results demonstrate the significance of leveraging predicate cardinalities to improve query planning.

The findings indicate that the "count" approach does not consistently lead to significant improvements in

query execution time, suggesting that counting triple pattern cardinalities and incorporating them by restarting join operations may not be sufficient for achieving effective adaptive querying.

Contrary to our expectations, which suggested that "count+index" would outperform both "count" and "index_card", we observe a decrease in performance when comparing "index" and "count+index". The inclusion of the "count" approach appears to have an adverse impact on the positive effects offered by the "index_card" approach. This finding implies that the synergy between these approaches is not favorable.

In conclusion, the results suggest that the "count" approach is not an effective strategy for adaptive querying, and its combination with the "index_card" approach in the "count+index" approach does not yield promising results. Instead, the findings emphasize that the "index_card" approach, which considers pre-generated predicate cardinalities, is the most effective approach for improving query performance in highly decentralized environments.

VIII. Conclusion

We compared diverse methods of acquiring cardinality information in order to identify the most suitable one to enhance the efficiency and performance of Link-Traversal-based Query Processing. This encompassed exploring the process of counting triple pattern occurrences during query execution on one hand, and examining the feasibility of utilizing pre-generated predicate cardinalities on the other hand.

We introduced three configurations. Our "index" configuration, fetching predicate cardinalities from the starting Solid pod of each query to guide join order decisions shows significant promise.

IX. Future Work

The concepts brought forward in this work are straightforward, and only form a first step towards improved query processing in heavily decentralized environments. This thesis identified many open questions and avenues for further research in the field, summarized in this section.

• By moving beyond the timeout-based method in the "count" approach, future approaches can allow a different restart timing for each join, potentially surpassing the performance demonstrated in this thesis. This avenue offers the potential to remain independent from user-generated information.

• Future work can combine counted cardinalities with those from the index in a more intricate way, provided they have reached a certain level of accuracy during query execution. Future work will have to investigate the point at which counted cardinalities can be deemed sufficiently accurate.

• Future work offers the potential to take indices into consideration from each pod encountered during query execution, this would lead to a more broadly applicable approach, as it would solve the performance limitations of query *discover-8*. Challenges that may arise in this context include obstructed query progress with queries that constantly traverse new pods, and intricacies in combining cardinalities from multiple sources in joins with entries drawing from various pods at once.

• The impact of our research on query response times, the time to generate a specific number of results, could be an interesting avenue for future work as query response time significantly influences the perceived performance of an information system.

• Using the right compression algorithms might considerably reduce the size of a file containing triple pattern cardinalities. This might make it feasible to use them, future research can investigate whether the impact on execution times is worth the resulting increased storage usage, potentially giving users the choice.

• Determining what patterns of deviation emerge between the real cardinalities and the approximations could form a comprehensive guide for future research, giving a clear idea of whether current research is on the right track.

References

[1] Ruben Taelman, Ruben Verborgh (2023) *Evaluation of Link Traversal Query Execution over Decentralized Environments with Structural Assumptions*

[2] Ghent University imec (2023) *Comunica: Link Traversal*

[3] Olaf Hartig (2011) *Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution*, 978-3-642-21033-4

[4] Solid team *Solid: your data, your choice.*

[5] Amol Deshpande, Zachary Ives, Vijayshankar Raman (2007) *Adaptive Query Processing*, Foundations and Trends in Databases

[6] Michael Schmidt, Georg Lausen (2010) *Foundations of SPARQL query optimization*

[7] Markus Stocker, Andy Seaborne, Abraham Bernsteinn, Christoph Kiefer, Dave Reynolds (2008) *SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation*

[8] Ruben Taelman, Joachim Van Herwegen, Miel Vander Sande, Ruben Verborgh (2018) *Comunica: A Modular SPARQL Query Engine for the Web*, The Semantic Web – ISWC 2018

[9] Ghent University imec (2023) *Comunica: A knowledge graph querying framework*

[10] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, Peter Boncz (2015) *The LDBC Social Network Benchmark: Interactive Workload*

[11] Sijin Cheng, Olaf Hartig (2022) *LinGBM: A Performance Benchmark for Approaches to Build GraphQL Servers (Extended Version)*

Queryprestaties in Gedecentraliseerde Omgevingen Verbeteren met Vooraf Gegenereerde Metadata en Adaptieve Queryplanning

Simon Van Braeckel
Supervisors: Dr. ir. Ruben Taelman, Jonni Hanski, Prof. dr. Pieter Colpaert, Prof. dr. ir. Ruben Verborgh

***Abstract*—Huidige decentralisatie-initiatieven zoals Solid bieden een hogere mate van decentralisatie dan ooit tevoren, een ontwikkeling waarop querytechnieken achterop lopen. Het gebrek aan vooraf beschikbare informatie voor query planners in deze omgevingen begint zijn problemen te tonen, aangezien geïntegreerde querytechnieken twee keer slechter presteren dan hun tegenhangers. Om deze kloof te overbruggen is er een groeiende behoefte aan betere queryplanning in deze context. We hebben uitbreidingen ontwikkeld voor de geïntegreerde Link-Traversal-gebaseerde aanpak om het verzamelen van planningsrelevante informatie, met name kardinaliteitsschattingen, en het bijwerken van het queryplan tijdens de uitvoering mogelijk te maken. We demonstrate an average performance improvement of 32.57% as a result of providing up front information to the query engine, showing that simple methods of providing up-front information provide a valuable basis for future work on query planning in decentralized environments. Door deze informatie vooraf te verstrekken aan de query engine bekomen we een gemiddelde prestatieverbetering van 32,57%, waaruit blijkt dat eenvoudige methoden voor het vooraf verstrekken van informatie een waardevolle basis vormen voor toekomstig werk op het gebied van queryplanning in gedecentraliseerde omgevingen.**

***Keywords*—Adaptive, Query Planning, Semantic Web, Linked Data, Link-Traversal, Join ordering**

I. Introduction

De afgelopen jaren is het web steeds meer gecentraliseerd geraakt. Om deze ontwikkeling tegen te gaan, zijn decentralisatie-initiatieven zoals Solid [4] populair geworden, die proberen de controle over gegevens terug te geven aan eindgebruikers. Momenteel ontbreken efficiënte zoektechnieken voor dergelijk sterk gedecentraliseerde contexten, omdat de meeste bestaande technieken zich richten op het gebruik in gecentraliseerde omgevingen [1].

Link-Traversal-gebaseerde Query Processing (LTQP) [2] is een aanpak die veelbelovend lijkt in deze context, maar die in de praktijk niet wijdverspreid wordt gebruikt vanwege prestatieproblemen, mede veroorzaakt door het gebrek aan vooraf beschikbare informatie over de gegevens. Deze thesis heeft tot doel deze prestatiebeperkingen aan te pakken door de bestaande techniek van zero-knowledge query planning [3] uit te breiden. Onze aanpak omvat het verzamelen van relevante planningsinformatie, zoals schattingen van kardinaliteit, en het bijwerken van het queryplan tijdens de uitvoering.

In de volgende sectie wordt het relevante verwante werk besproken, waarna onze volledige oplossing wordt gepresenteerd. Vervolgens leggen we de use case uit waarop de experimenten zijn gebaseerd. De resultaten worden daarna gepresenteerd. Ten slotte worden enkele afsluitende opmerkingen gemaakt, gevolgd door suggesties voor verder werk.

A. Link traversal query processing

Link-Traversal-gebaseerde Query Processing (LTQP) [2] is een queryparadigma dat werd geïntroduceerd als een manier om het Web van Linked Data te bevragen zonder dat het eerst in een centrale locatie geïndexeerd hoeft te worden [1]. Binnen LTQP zijn de bronnen niet van tevoren bekend. In plaats daarvan vinden ontdekking van gegevensbronnen plaats tijdens de queryuitvoering.

In vergelijking met traditionele queryverwerking kunnen geïntegreerde aanpakken zoals LTQP niet vertrouwen op algoritmen die de query optimaliseren voor de uitvoering omdsat die veelal voorafgaande statistieken van datasets vereisen. In plaats daarvan maakt het gebruik van een zero-knowledge query plan techniek [3] om triple-patronen te ordenen op basis van specifieke heuristieken voor Link Traversal.

Volgens gerelateerd onderzoek [1] ligt het belangrijkste werkpunt binnen LTQP vooral op het gebied van het queryplan in plaats van de ontdekking van relevante documenten.

1) Evaluation of Link Traversal Query Execution over Decentralized Environments with Structural Assumptions: Dit gerelateerde artikel presenteert een vergelijking tussen een geïntegreerde aanpak en een two-phase aanpak van query-uitvoering. De two-phase aanpak fungeert als een theoretisch geval en maakt gebruik van een orakel dat alle query-relevante links levert, waardoor het gebruik kan maken van traditionele pre-executie optimaliseeralgoritmen [6], [7]. De two-phase aanpak is gemiddeld twee keer sneller dan de geïntegreerde aanpak, wat kan worden toegeschreven aan zijn vermogen om traditionele queryplanning uit te voeren, waarbij onder andere gebruik wordt gemaakt van kardinaliteitsschattingen. Dit aanzienlijke potentieel benadrukt de behoefte aan effectievere queryplanningstrategieën. Ons werk streeft ernaar om deze kloof te overbruggen door het queryplan tijdens de uitvoering aan te passen.

B. Comunica

Comunica [9] is een adaptief query-engine framework dat het mogelijk maakt om een query-engine samen te stellen die geïntegreerde queryplanning en -uitvoering doet. De implementaties die in dit proefschrift zijn bijgedragen zijn gemaakt als uitbreidingen op Comunica.

II. Related Concepts

A. The Query Process

Query-verwerking omvat traditioneel gezien drie belangrijke stappen: parsing, planning en uitvoering. De eerste stap, parsing, is waar de SPARQL-query wordt omgezet in een algebraïsche expressie. De tweede fase, de planningsfase, transformeert de algebraïsche expressie tot een queryplan. Een queryplan is een stapsgewijze handleiding die de volgorde van bewerkingen specificeert die nodig zijn om de gewenste gegevens op te halen. Nadat het queryplan is verkregen uit de planningsfase wordt het uitgevoerd in de derde en laatste fase van het queryproces.

De scheiding van queryplanning en -uitvoering leidt tot suboptimale uitvoeringstijden in sterk gedecentraliseerde omgevingen, waarbij optimalisatie van het queryplan wordt belemmerd omdat het afhankelijk is van vooraf beschikbare informatie over de gegevens.

Zoals vermeld in de inleiding, is LTQP een belangrijke techniek binnen deze context. Het gaat om het feit dat relevante gegevensbronnen voorafgaand aan de query-uitvoering onbekend zijn door de integratie van bronontdekking en uitvoering. Het is echter aangetoond dat de eenvoud binnen LTQP kan leiden tot suboptimale queryplannen, waardoor het twee keer zo langzaam is als een oplossing waarbij het queryplan optimaal is [1].

Om deze kloof te overbruggen, beschrijven we in dit proefschrift het gebruik van adaptieve queryverwerkingstechnieken, die gebruik maken van runtime-feedback, en zo het plan of de planningruimte [5] aanpassen tijdens de uitvoering, waarbij de planning- en uitvoeringsfasen nauw met elkaar verweven zijn. Adaptieve technieken zijn veelbelovend in dit geval, omdat door het queryplan te wijzigen naarmate meer informatie over de gegevens beschikbaar wordt het queryplan dichter bij het ideaal komen.

We introduceren adaptiviteit in de vorm van de herordening van entries voor join-operaties. Een join-operatie verwijst naar het proces van het combineren van triple patronen binnen een query. Triple patronen zijn combinaties van waarden of variabelen voor subject, object en predicaat, die een specifieke structuur opleggen aan triples. Deze triple patronen bevatten vaak gemeenschappelijke variabelen, waarbij bindings voor een variabele van het ene triple patroon worden vergeleken met bindings voor dezelfde variabele in een ander triple patroon. We kiezen de juiste volgorde van join entries op basis van hun kardinaliteit, dat wil zeggen de hoeveelheid bindings aanwezig in de entries.

Door zorgvuldig de volgorde te kiezen waarin de entries worden samengevoegd, kunnen we de generatie van tussenresultaten verminderen en onnodig werk vermijden, wat leidt tot snellere query-uitvoering.

Helaas zijn de exacte kardinaliteiten van join entries in ons geval niet bekend, dus werken we met geschatte kardinaliteiten. Om deze geschatte kardinaliteiten te gebruiken om join entries te ordenen, gaan we ervan uit dat het het beste is om eerst de entries met de kleinste kardinaliteiten samen te voegen. Deze leiden waarschijnlijk tot een klein aantal resultaten, wat het aantal bindings in latere joins vermindert en op zijn beurt de totale benodigde inspanning vermindert.

Aangezien join-operaties plaatsvinden op triple patronen, stelt het kennen van de exacte kardinaliteit van een triple patroon de query-engine in staat om nauwkeurige voorspellingen te doen met betrekking tot tussenresultaten tijdens join-operaties. Hierdoor zijn TP-kardinaliteiten een ideaal criterium om join entries te sorteren en de query-prestaties te optimaliseren.

III. Use Case

We hebben onze oplossingen getest met de Solidbench [1] benchmark, die voortbouwt op de gevestigde Social Network Benchmark (SNB) [10], [11].

Deze benchmark modelleert een query-workload en gegevensstructuur zoals men zou vinden in een sociaalnetwerkscenario, opgesplitst om gedecentraliseerde werklasten te simuleren die vergelijkbaar zijn met die in Solid. Hoewel deze benchmark de context van Solid simuleert, zijn de oplossingen in deze thesis generaliseerbaar naar andere Linked Data-omgevingen.

IV. Solution

In deze sectie leiden we onze drie verschillende configuraties in, deze hebben als doel om query executie in gedecentraliseerde omgevingen te optimaliseren.

A. Counting Triple Pattern Cardinalities & Timeout

Ondanks het belang van informatie over de kardinaliteiten van join entries bij het maken van geïnformeerde beslissingen over de volgorde van joins, is dergelijke informatie veelal niet beschikbaar in een gedistribueerde, gedecentraliseerde context.

Om deze beperking aan te pakken, hebben we een methode geïmplementeerd om de voorkomens van Triple Patterns (TP) te tellen tijdens executie van de query, dit combineren we met een timeout mechanisme. Het timeout mechanisme herstart de join operatie na een bepaalde tijd, waardoor er potentieel meer nauwkeurige kardinaliteitsinformatie beschikbaar is.

Het belangrijkste voordeel van deze aanpak is diens onafhankelijkheid van het beschikbaar zijn van vooraf gegenereerde gegevens. Deze onafhankelijkheid vergroot de flexibiliteit en toepasbaarheid van de queryengine.

Het tweede voordeel is dat deze aanpak het gebruik van TP-kardinaliteiten mogelijk maakt zonder overmatig schijfruimtegebruik te vereisen om deze kardinaliteiten op te slaan in een bestand.

B. Predicate Cardinality File

Hoewel TP-kardinaliteiten waardevol zijn is het creëren van een bestand met de kardinaliteit van elk mogelijk Triple Pattern in vele real-world scenario's onhaalbaar vanwege de enorme hoeveelheid mogelijke combinaties in een Solid-pod. Voor kleinere Solid-pods ligt dit mogelijks anders, vooral als de geschikte compressiealgoritmen worden gebruikt. Dit wordt in meer detail besproken in onze sectie over toekomstig werk.

We kiezen voor predicaat kardinaliteiten als een tussenoplossing en gebruiken de kardinaliteit van het predicaat van elke join entry als een aanpak van de TP-cardinaliteit. We onderzoeken of deze aanpak voldoende nauwkeurig is om prestatieverbeteringen te realiseren.

Binnen deze configuratie verkrijgen we predicaat kardinaliteiten aan het begin van de query-uitvoering. Nadat de kardinaliteiten zijn verkregen, wordt de daaropvolgende uitvoering voortgezet zoals gebruikelijk, waarbij de kardinaliteiten worden gebruikt om toekomstige join entries te sorteren. We beoordelen de impact van directe toegang tot de predicaat kardinaliteiten.

C. Combining Predicate Cardinality File and Timeout

De derde aanpak combineert predicaat-kardinaliteiten met TP-kardinaliteiten. We beginnen onze uitvoering door predicaat-kardinaliteiten op te halen uit de Solid-pod zoals in de tweede methode. Tijdens de query-uitvoering, worden de getelde Triple Pattern kardinaliteiten gebruikt wanneer predicaat-kardinaliteiten niet beschikbaar zijn voor een bepaalde join entry.

V. Experimental Design

We hebben zes verschillende aanpakken geëvalueerd, waaronder de baseline aanpak en een variatie daarop. De geëvalueerde aanpaken zijn als volgt: nosep

- **base_zero**: zero-knowledge query planning [3]. Hier wordt ook naar verwezen als de baseline aanpak.
- **base_card**: Baseline aanpak, aangepast zodat het een sorteeralgoritme op basis van kardinaliteiten gebruikt. Deze aanpak werd vergeleken met de queryplanning aanpak op basis van zero-knowledge om de effectiviteit van zero-knowledge te beoordelen.
- **index_card**: Uitbreiding op de baseline aanpak waarbij gebruik wordt gemaakt van predicaat-kardinaliteiten zoals beschreven in Sectie IV-B.
- **index_zero**: Index_card aanpak, uitgerust met de zero-knowledge sorteerwijze. Deze aanpak wordt vergeleken met de index_card om de effectiviteit van het sorteren op basis van predicaat-kardinaliteiten te verifiëren.
- **count**: Uitbreiding op de baseline aanpak die Triple Patterns telt en joins opnieuw start, zoals beschreven in Sectie IV-A.
- **count+index**: Count aanpak gecombineerd met index_card aanpak, zoals beschreven in Sectie IV-C.

We hebben de SolidBench [1] benchmark gebruikt zoals beschreven in Hoofdstuk III. We hebben echter gemerkt dat elk van de complexe queries resulteerde in een timeout. Als gevolg daarvan hebben we deze complexe queries uit onze analyse weggelaten. We hebben elke overgebleven query drie keer uitgevoerd en gemiddelden van metingen berekend om betrouwbare resultaten met minder ruis te verkrijgen. Om te voorkomen dat de query-uitvoering te lang duurde, hebben we een maximale duur van 1 minuut ingesteld.

Onze experimenten zijn uitgevoerd op een 64-bits Pop!_OS 22.04 LTS-machine met een 8-core Intel i7-10510U 1,80 GHz CPU en 16 GB RAM. Zowel de server die de Solid-data levert als de client die de queries uitvoert, werden op dezelfde machine uitgevoerd.

We hebben de uitvoeringstijden gemeten voor elke query en configuratie; de voltallige resultaten zijn te vinden in de volledige scriptie. In de volgende sectie bespreken we enkel onze samengevatte resultaten.

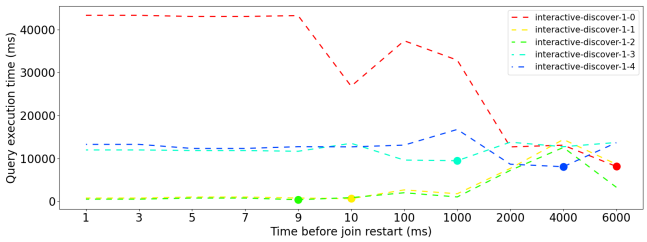


Figure 1: Optimale timeoutwaarde en uitvoeringstijd voor query discover 1 in configuratie "count"

VI. Results

A. Subsetting the query set

We hebben bepaalde querycategorieën geïdentificeerd die op basis van hun kenmerken niet verder in de analyse zijn opgenomen. Naar deze subset wordt verder verwezen als de representatieve subset.

• Query *discover-8* vertoonde langere uitvoeringstijden als de index aanpak werd gebruikt. De oorzaak hiervan is dat deze query tijdens de uitvoering toegang benodigd tot meerdere Solid pods. Onze implementatie was hier niet specifiek op afgestemd. Hierdoor zouden de predicaat-cardinaliteiten van de startpod nog steeds worden gebruikt om beslissingen over de joinvolgorde te sturen, terwijl de gegevens afkomstig zijn van de nieuwe pod. We sluiten deze query uit van toekomstige evaluaties en vermelden de belangrijke use case die ze vertegenwoordigt als ons voornaamste aandachtspunt voor toekomstig onderzoek.

• Queries *discover-3*, *discover-4* en *short-4* vertoonden uitzonderlijk lage uitvoeringstijden in de baseline configuratie. Als gevolg hiervan kunnen kleine, insignificante variaties in uitvoeringstijd resulteren in aanzienlijke procentuele veranderingen. Het opnemen van deze queries in onze berekeningen zou de gemiddelden sterk verstoord hebben.

• De baseline aanpak ondervond timeouts in queries *short-2*, *short-3* en *short-7*, wat de vergelijking met deze aanpakken compliceerde.

• Voor queries *short-1*, *short-3* en *short-5* lukte het onze "count" en "count+index" aanpakken niet om het juiste aantal resultaten op te halen.

B. Discussion

1) There is no globally optimal timeout value: In onze op tellen gebaseerde benadering maken we gebruik van een timeoutwaarde om join-operaties opnieuw te starten nadat een bepaalde hoeveelheid tijd is verstreken. Figuur 1 laat zien hoe de uitvoeringstijd van de "count" methode evolueert afhankelijk van deze timeoutwaarde.

Dit resultaat benadrukt het ontbreken van een universeel optimale timeoutwaarde voor alle queries. In feite is het gebruik van een enkele timeoutwaarde voor alle join-operaties binnen een enkele query mogelijks onrealistisch, gezien de verschillende aard van elke join-operatie. In plaats de globaal optimale timeoutwaarde te proberen vinden evalueren we de uitvoeringstijd van deze configuratie met behulp van de timeoutwaarde die de beste uitvoeringstijd oplevert per query. In ons toekomstige werk bespreken we hoe deze aanpak kan worden verbeterd.

2) Interference in timeout measurements: Een aanzienlijk aantal queries vertoont ideale timeoutwaarden die de uitvoeringstijd overschrijden. In deze gevallen wordt er geen enkele join opnieuw gestart. We kunnen concluderen dat deze specifieke gevallen geen voordeel halen uit het opnieuw starten van de join, wat impliceert dat de getelde kardinaliteiten nadelige effecten hebben op de join-volgorde, of dat de overhead van het opnieuw starten significant hoog ligt.

Deze conclusie kan echter aanzienlijk beïnvloed zijn door de uniforme toepassing van dezelfde timeoutwaarde op alle joins. Bijvoorbeeld, een query met een join die optimaal een timeout van 10ms zou hebben, kan betere uitvoeringstijden vertonen wanneer een andere globale timeoutwaarde wordt gebruikt, enkel omdat de andere join-operaties geen baat hebben bij het opnieuw starten na 10ms. Dit onderdeel vereist een grondiger onderzoek.

3) Analysis using the representative subset: Deze tabel vergelijkt verschillende aanpakken met de baseline aanpak. De eerste twee kolommen tonen het percentage queries waarvoor elke aanpak uitvoeringstijden vertoont die minstens 10% hoger of lager zijn dan de baseline, terwijl de laatste kolom de gemiddelde verbetering in uitvoeringstijd weergeeft als een fractie van de prestaties van de baseline aanpak.

	% van queries		
	10% lager	10% hoger	gemiddelde verbetering
index_zero	0,00%	5,33%	-6,79%
index_card	20,00%	1,33%	32,57%
count	5,33%	5,33%	0,23%
count+index	14,67%	1,33%	29,39%

Table I: Percentage van de queries uit representatieve subset met een uitvoeringstijd die minstens 10% lager (resp. 10 hoger) is dan de basislijn

Figuur 2 laat zien in hoeveel queries elk van de aanpakken de snelste uitvoeringstijd had.

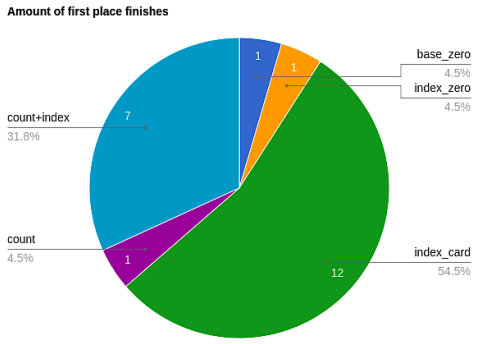


Figure 2: Aantal queries waarin configuraties het snelst zijn

De resultaten met behulp van de representatieve subset (Sectie VI-A) tonen aan dat de "index_zero" aanpak, resulterend in een gemiddelde prestatiedaling van 6,79%, minder effectief is in vergelijking met de baseline aanpak. Het bewijs ondersteunt het belang van het sorteren van join entries met kardinaliteitsinformatie om betere queryplannen te realiseren. Toekomstig werk kan bepalen of de negatieve invloed van de aanpak op de prestaties te wijten is aan de overhead in onze implementatie van indexontdekking, of aan de resultaten van de join entry sorteringen.

De "index_card" aanpak blijkt de meest effectieve van de geëvalueerde technieken. Gemiddeld presteert deze beter dan de baseline "index_zero" aanpak met een aanzienlijke 32,57%. De resultaten tonen het belang aan van het benutten van predicaatkardinaliteiten om queryplanning te verbeteren.

De bevindingen geven aan dat de "count" aanpak niet leidt tot significante verbeteringen in de uitvoeringstijd van queries, wat suggereert dat het tellen van kardinaliteiten van triple patronen en het opnieuw starten van join-operaties mogelijks niet voldoende is voor effectieve adaptieve querying.

In tegenstelling tot onze verwachting, die suggereerde dat "count+index" zowel "count" als "index_card" zou overtreffen, zien we een afname in prestaties in "count+index". De opname van de "count" aanpak lijkt een negatieve invloed te hebben op de positieve effecten die de "index_card" aanpak biedt.

Als conclusie, uit de resultaten blijkt dat de "index_card" aanpak, die pre-generatie predicaatkardinaliteiten in overweging neemt, de meest effectieve aanpak is voor het verbeteren van de query-prestaties in sterk gedecentraliseerde omgevingen.

VII. Conclusion

We hebben verschillende methoden voor het verkrijgen van cardinaliteitsinformatie vergeleken om de meest geschikte methode te identificeren om de efficiëntie en prestaties van Link-Traversal-gebaseerde Query Processing te verbeteren. Deze verkenning omvatte zowel het onderzoeken van het tellen van het aantal voorkomens van Triple Patterns tijdens de query-uitvoering als het evalueren van de haalbaarheid van het gebruik van vooraf gegenereerde predicaatkardinaliteiten.

We hebben drie configuraties geïntroduceerd. Onder deze configuraties toont onze "index" configuratie, die predicaatcardinaliteiten ophaalt vanuit de initiële Solid-pod van elke query om de volgorde van joins te begeleiden, veelbelovende resultaten.

VIII. Future Work

De concepten die in dit werk worden gepresenteerd, zijn eenvoudig en vormen slechts een eerste stap richting verbeterde queryverwerking in sterk gedecentraliseerde omgevingen. Dit proefschrift heeft veel open vragen en onderzoekspaden geïdentificeerd, die worden samengevat in deze sectie.

- Door verder te gaan dan de timeout-gebaseerde methode in de "count" benadering, kunnen toekomstige benaderingen verschillende momenten van herstarten voor elke join toestaan. We verwachten dat het mogelijk is om het prestatieniveau gedemonstreerd in dit proefschrift te overstijgen. Deze benadering biedt het potentieel om onafhankelijk te blijven van gebruikersgegenereerde informatie.
- Toekomstig werk kan getelde kardinaliteiten op een meer complexe manier combineren met die van de index, op voorwaarde dat ze een bepaald niveau van nauwkeurigheid hebben bereikt. Toekomstig onderzoek zal manieren moeten vinden om te bepalen of getelde kardinaliteiten als voldoende nauwkeurig kunnen worden beschouwd.
- Toekomstig werk biedt de mogelijkheid om indices in overweging te nemen van elke pod die tijdens de query-uitvoering wordt aangetroffen. Dit zou leiden tot een breder toepasbare benadering, omdat het de prestatiebeperkingen van query *discover-8* zou oplossen. Uitdagingen die in deze context kunnen

ontstaan, omvatten belemmerde query uitvoering bij query's die voortdurend nieuwe pods ontdekken, en complexiteiten bij het combineren van kardinaliteiten uit meerdere bronnen wanneer een join operatie entries uit meerdere pods gebruikt.

• Het effect van ons onderzoek op de responstijden van queries, de tijd die het vergt om een specifiek aantal resultaten te genereren, kan een interessante richting zijn voor toekomstig onderzoek, aangezien de responstijd van queries een aanzienlijke invloed heeft op de ervaren prestatie van een informatiesysteem.

• Het gebruik van de juiste compressiealgoritmen kan de omvang van een bestand met kardinaliteiten van Triple Patterns aanzienlijk verminderen. Dit zou het mogelijk kunnen maken om ze te gebruiken. Toekomstig onderzoek kan onderzoeken of de impact op uitvoeringstijden de resulterende toegenomen opslagruimte waard is, mogelijks kan de keuze tussen predikaat en Triple Pattern kardinaliteiten bij de gebruikers worden gelaten.

• Het bepalen van de afwijkingenpatronen tussen de echte kardinaliteiten, zoals tegengekomen tijdens de uitvoering, en de benaderingen kan een gedetailleerde gids vormen voor toekomstig onderzoek. Door de echte kardinaliteiten te meten kan men een duidelijk beeld geven of het huidige onderzoek op de juiste weg is.

References

[1] Ruben Taelman, Ruben Verborgh (2023) *Evaluation of Link Traversal Query Execution over Decentralized Environments with Structural Assumptions*

[2] Ghent University imec (2023) *Comunica: Link Traversal*

[3] Olaf Hartig (2011) *Zero-Knowledge Query Planning for an Iterator Implementation of Link Traversal Based Query Execution*, 978-3-642-21033-4

[4] Solid team *Solid: your data, your choice.*

[5] Amol Deshpande, Zachary Ives, Vijayshankar Raman (2007) *Adaptive Query Processing*, Foundations and Trends in Databases

[6] Michael Schmidt, Georg Lausen (2010) *Foundations of SPARQL query optimization*

[7] Markus Stocker, Andy Seaborne, Abraham Bernsteinn, Christoph Kiefer, Dave Reynolds (2008) *SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation*

[8] Ruben Taelman, Joachim Van Herwegen, Miel Vander Sande, Ruben Verborgh (2018) *Comunica: A Modular SPARQL Query Engine for the Web*, The Semantic Web – ISWC 2018

[9] Ghent University imec (2023) *Comunica: A knowledge graph querying framework*

[10] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, Peter Boncz (2015) *The LDBC Social Network Benchmark: Interactive Workload*

[11] Sijin Cheng, Olaf Hartig (2022) *LinGBM: A Performance Benchmark for Approaches to Build GraphQL Servers (Extended Version)*

Table of contents

List of Figures

List of Tables

Source Code

The implementation presented in this work can be found on different branches in the repository https://github.com/simonvbrae/comunica-feature-link-traversal.

- baseline: https://github.com/simonvbrae/comunica-feature-link-traversal/tree/feature/adaptive-join-cardinalities-baseline
- baseline cardinality sort: https://github.com/simonvbrae/comunica-feature-link-traversal/tree/feature/adaptive-join-cardinalities-baseline-card
- index: https://github.com/simonvbrae/comunica-feature-link-traversal/tree/feature/adaptive-join-cardinalities-index
- count: https://github.com/simonvbrae/comunica-feature-link-traversal/tree/feature/adaptive-join-cardinalities-count
- count+index: https://github.com/simonvbrae/comunica-feature-link-traversal/tree/feature/adaptive-join-cardinalities-count-index

The code for executing the experiments and plotting their results can be found at https://github.com/simonvbrae/LTQP_cardinalities_experiments.

1

Preface

1.1 Introduction

Today, the majority of data on the Web is flowing towards isolated data silos, which are in the hands of large companies. This ever-increasing siloization of data leads to a host of challenges including issues with cross-silo interoperability, vendor lock-in, privacy concerns, and individuals' lack of control over their own data. As data flows towards isolated data silos, users find themselves constrained by limited portability and interoperability between services, motivated by companies' prioritization of their own product over facilitating data portability with competitors. As a result, this increased centralization stifles technological development and restricts user autonomy.

The implications of this centralization extend beyond the realm of user experience and into broader societal concerns. Companies like Facebook and Google, serving as identity providers for various applications, amass vast amounts of data pertaining to individuals' social activities and service consumption [18]. The information collected by these companies is exploited to shape consumer behavior by delivering targeted advertisements. Moreover, this data has been used to target tailored advertisements for presidential campaigns catered to specific demographic groups, building psychographic profiles [27] of voters and targeting ads to influence voting behavior [19].

To address these issues and empower users, decentralization initiatives like Solid [24] have gained popularity. Solid offers an alternative by allowing users to store their data in personal data vaults, known as pods, distributed across the web. This decentralized storage model gives individuals full control to determine which applications can interact with their data.

These personal data vaults form Knowledge Graphs [17], which are collections of Linked Data documents [3] containing RDF triples [6]. Building applications on top of this decentralized Knowledge Graph presents significant technical challenges. Traditional centralized aggregation prior to query

processing is not feasible due to legal restrictions, and existing federated querying techniques are ill-suited [26] for the large-scale distribution of data in environments like Solid.

Currently, there is a lack of efficient querying techniques specifically designed for environments like Solid, which are significantly more decentralized than ever before. Link-Traversal-based Query Processing (LTQP) [11] shows promise as an approach for heavily decentralized environments such as Solid. LTQP enables querying in a context where documents are being discovered during query execution, following the principles of the Linked Data paradigm.

Despite its potential, LTQP has not been widely adopted in practice due to performance concerns related to lack of upfront information about the data. This thesis aims to address these performance limitations by gathering planning-relevant information, such as cardinality estimates, and updating the query plan accordingly during execution. We focus our analysis and evaluation on the Solid ecosystem, but the concepts and techniques developed in this work may have broader applicability to other decentralization initiatives.

1.2 Outline

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of related Semantic Web technologies, while Chapter 3 delves into the intricacies of query processing. In Chapter 4, we discuss existing related work and compare it to our work. Moving forward, Chapter 5 introduces a use case that serves as the basis for testing our proposed approaches. We then proceed to Chapter 6, where we formally present the problem statement, research questions, and hypotheses. Chapter 7 details the architecture of our different implementations. We evaluate these approaches in Chapter 8, followed by concluding remarks and a discussion of future work in Chapter 9.

2

Overview of Relevant Semantic Web Technologies

In this chapter, we give important background information on technologies related to this work. Firstly, we discuss the Semantic Web, after which we give a brief overview of the Resource Description Framework and SPARQL. Finally, we discuss the Linked Data principles and the Solid specifications.

2.1 The Semantic Web

The current generation of search engines primarily functions as location finders[4], searching for the location of text-based information on the web but lacking the capability to fully understand the meaning and context behind the data they retrieve.

One promising solution to address this limitation is the integration of Artificial Intelligence (AI) into search engines. By incorporating AI technologies, such as computational linguistics, search engines can enhance their ability to comprehend and interpret the content they index, leading to more sophisticated and accurate information retrieval.

An alternative approach to enable more intelligent information retrieval is through the Semantic Web initiative. This initiative aims to represent web content in a format that is easily machine-processable. By employing standardized data models and intelligent techniques, the Semantic Web seeks to add semantic meaning to the web, making it more structured and maintaining or increasing it's heavy interconnectedness.

RDF, Linked Data, and SPARQL form the foundational technologies that support the concept of the Semantic Web. By utilizing these technologies, the Semantic Web aims to create a web of intercon-

nected knowledge, where data is not only human-readable but also machine-processable.

2.2 Resource Description Framework

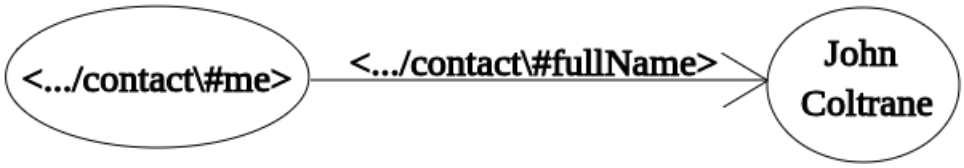
RDF (Resource Description Framework) [6] is a framework for representing data on the web. It extends the URI-based linking structure of the web, enabling the representation of relationships between different entities. In RDF, these relationships are captured as *triples*, which consist of a subject, predicate, and object. In their simplest form, the subject, object, and predicate are represented as Internationalized Resource Identifiers (IRIs). However, the object can also be a data literal, allowing the mixing of structured and semi-structured data.

RDF data can be imagined as a table with three columns: subject, predicate, and object, similar to a relational database in SQL. This straightforward and flexible data model possesses significant expressive power, capable of representing complex situations and relationships while maintaining an appropriate level of abstraction.

For example, following triple can be used to express the information that there is a Person identified by the URI http://www.w3.org/contact#me, whose name is John Coltrane.

<http://www.w3.org/contact#me> <http://www.w3.org/contact#fullName> "John Coltrane" .

This linking structure forms a directed, labeled graph, where the edges represent named links between two resources, represented by the graph nodes. This graph view is a simple mental model for understanding RDF's structure.



2.3 SPARQL

SPARQL [1] is the W3C-standardized query language for retrieving and manipulating data stored in RDF format. With SPARQL query engines, users gain the ability to search the Web of Data or any RDF

database to uncover relevant triples that match their queries.

SPARQL is a declarative language, similar to SQL, where the focus lies on defining what data to find rather than the precise method of finding it. This declarative nature of SPARQL allows for a clear separation between query specification and execution. Consequently, the execution process becomes independent of the query engine's implementation, fostering a higher degree of flexibility and simplicity.

2.4 Linked Data

Linked Data [3], one of the core pillars of the Semantic Web, is a set of design principles that facilitate the sharing of machine-readable interlinked data on the Web. Introduced by Sir Tim Berners-Lee in 2006, these principles lay the foundation for a more interconnected and accessible data landscape. Let's explore the four fundamental principles of Linked Data:

1. Use URIs as names for things. Uniform Resource Identifiers (URIs) serve as unique identifiers for different entities, allowing us to distinguish between items or recognize equivalent entities across various datasets. By providing a standardized naming mechanism, URIs foster interoperability and data integration.

2. Use HTTP URIs for easy look-up. Employing HTTP URIs enables straightforward resource retrieval through the Hypertext Transfer Protocol (HTTP). This means that when items are identified using URIs combined with HTTP, they become easily discoverable and accessible, making the process of publishing and incorporating data into the global data space more efficient.

3. Provide useful information when URIs are looked up. When someone queries a URI, it should return useful information about the entity it represents. This is typically achieved using standardized data representation formats like RDF (Resource Description Framework) and SPARQL (SPARQL Protocol and RDF Query Language). These standards ensure that data is semantically meaningful and can be effectively processed by machines.

4. Include links to other URIs for further exploration. Just like the hypertext web, Linked Data relies on interlinking resources through URI references. By linking new information to existing resources, we create a richly interconnected network of data with immense potential for data exploration. This interlinking maximizes data reuse and facilitates the discovery of new relationships between different entities.

The benefits of employing Linked Data are substantial. It dismantles the barriers between data silos and allows integration of diverse data formats from various sources. As a result, data integration and exploration become more straightforward and efficient. Additionally, the linking of disparate sources and formats enables the inference of new knowledge from existing facts. By leveraging Linked Data, organizations can connect their proprietary knowledge with open-world or specialized knowledge.

2.5 Solid

Solid is a set of specifications that enables individuals to securely store their data in decentralized data stores called Pods. These Pods act as personal web servers and provide users with control over their data. With Solid, any type of information can be stored in a Pod. [24]

One of the key features of Solid is that it puts users in control of their data access. Solid enables users to decide what data to share and with whom, whether it be individuals, organizations, or applications. This control also extends to the ability to revoke access to data at any time. By using standard, open, and interoperable data formats and protocols, Solid facilitates seamless interoperability between applications.

In the introduction, we mentioned the challenges posed by isolated data silos, centralization, and the resulting vendor lock-in. These issues not only raise concerns about privacy but also pose threats to our political and personal freedom. Fortunately, Solid is a technology that holds the potential to mitigate these threats.

To illustrate the benefits of Solid, let's consider the example of messaging. Imagine your messages are stored in your personal Solid pod by Facebook, and WhatsApp also adopts Solid for its messaging platform. In this scenario, users have the freedom to port their messages from Facebook to WhatsApp seamlessly. Furthermore, the accessibility of their own data empowers users to write their own conversion script, even if the two platforms use different data formats.

By enabling such platform choices and data portability, Solid encourages competition in markets currently dominated by monopolies. Fostering a more transparent and user-centric ecosystem.

Despite the advantages Solid may bring, it presents novel challenges with querying. The main challenge arises as a result of its large-scale decentralization. Compared to federated approaches that deal with a relatively small number of sources, usually around 10, Solid involves a vast number of sources, potentially reaching millions [26]. This exponential increase in decentralization poses

unique challenges for query processing.

Another notable complication that has been identified is due to the lack of statistics available prior to query execution. The absence of statistics makes it difficult to perform accurate query planning. Instead, query engines rely on heuristics to plan the queries effectively [14]. However, heuristics may not always yield optimal query plans, leading to suboptimal execution times.

To overcome the limitations of heuristic-based query planning and improve query performance in Solid environments, there is a need for adaptive query planning approaches (Section 4.3), which we aim to address in this thesis.

3

Query Processing

3.1 Overview of Query Processing

Query processing is a fundamental aspect of executing SPARQL queries over Knowledge Graphs. It involves three key steps: parsing, planning, and execution.

The first step, parsing, is where the SPARQL query string is transformed into an algebraic expression. The second phase, the planning phase, transforms the algebraic expression into a query plan. A query plan is a step-by-step manual that specifies the sequence of operations required to retrieve the desired data. The planning phase is essential because queries are declarative, meaning they state *what* needs to be done, but not *how* it should be done. Multiple ways exist to execute the same query, and the planning phase determines the high-level approach used during execution.

After obtaining the query plan from the planning phase, there are still many ways to execute it. This variability is part of the query execution phase, which involves carrying out the steps specified in the query plan. While this phase is crucial for actualizing the query plan, this thesis's primary focus is contained within the planning phase.

It is worth noting that in adaptive systems (Section 4.3), such as the approach proposed in this thesis, query execution and query planning are tightly intertwined. The adaptivity aspect allows for dynamic changes to the query plan during execution, providing opportunities for optimization based on ongoing data discovery and processing.

This thesis aims to incorporate adaptivity into the query process. By re-ordering join entries during execution, thus changing the query plan adaptively, we seek to reduce query execution times. The sequence in which joins are executed heavily influences the overall query execution time. In this chapter, we delve into join operations and emphasize the importance of their proper ordering to

optimize query execution efficiency.

3.2 Join Operations

In executing a query, the declarative query first gets converted to a query plan, which serves as a blueprint or roadmap for the query engine to follow. The query plan specifies the order of executing operations, such as filtering, sorting, and joining, to efficiently retrieve and combine the relevant data.

As the query plan is often riddled with join operations, they play a crucial role in query execution. There are different types of logical join operations, including inner join, optional join, and minus join [9]. Each logical join has different semantics, implying distinct meanings and functioning methods.

Each logical join type can be implemented using multiple physical join algorithms. However, the choice of physical join algorithms is the responsibility of the query engine, and we will not delve further into physical join algorithms for the purpose of this thesis.

As a result of the prominence of join operations within the query plan, the efficiency of the query process significantly depends on how the join operations are planned. In this work we will focus on enhancing query performance by improving the planning of join operations through adaptive techniques and leveraging pre-generated metadata. This section provides a brief introduction to the key concepts and considerations related to join operations.

Consider the following SPARQL query:

```
SELECT * WHERE {
  ?start <ex:p1> ?p.
  ?p <ex:p2> ?end.
}
```

This query requires two Triple Patterns to be joined.

1. ?start <ex:p1> ?p.
2. ?p <ex:p2> ?end.

A query engine can represent this as two join entries, each with bindings for specific variables:

- Join entry 1 with bindings for variables ?start and ?p
- Join entry 2 with bindings for variables ?p and ?end

Joining these two entries results in an intermediary join operation that produces bindings for the variables ?start, ?p, and ?end. The bindings in this intermediary operation will contain all existing combinations of these variables based on the triples present in the two underlying join entries.

As an illustrative example of the join operation with example data, let's consider the following bindings for the two join entries:

- Join entry 1:
  { start: "ex:s1"; p: "ex:p1" }
  { start: "ex:s2"; p: "ex:p2" }
  { start: "ex:s3"; p: "ex:p3" }
- Join entry 2:
  { p: "ex:p1", end: "ex:o1" }
  { p: "ex:p1", end: "ex:o2" }
  { p: "ex:p3", end: "ex:o3" }

If we determine the possible combinations of these join entries following the inner join semantics, then we will obtain the following bindings:

- Joined bindings:
  { start: "ex:s1"; p: "ex:p1"; end: "ex:o1" }
  { start: "ex:s1"; p: "ex:p1"; end: "ex:o2" }
  { start: "ex:s3"; p: "ex:p3"; end: "ex:o3" }

Note that the second binding of the first join entry does not appear in the final results. This is caused by the value "ex:p2" for the variable ?p not occurring in the second join entry's bindings.

3.3 Significance of Join Ordering in Query Execution

Building on the previous example, this section illustrates the crucial role of join ordering in optimizing the efficiency of query execution in a Linked Data context. This example aims to illustrate that the order in which join operations are performed can significantly alter how the data is processed, without necessarily having an impact on result completeness or correctness. This change can influence the amount of computational effort required and the size of intermediate results generated during the execution of a query.

Consider the following SPARQL query, which consists of three Triple Patterns that need to be joined:

```
SELECT * WHERE {
  ?start <ex:p1> ?p.
  ?p <ex:p2> ?q.
  ?q <ex:p3> ?end.
}
```

Assume the following bindings for the join entries:

- Join entry 1:
  { s: "ex:s1"; p: "ex:p1" }
  { s: "ex:s2"; p: "ex:p2" }
  { s: "ex:s3"; p: "ex:p3" }
- Join entry 2:
  { p: "ex:p1", q: "ex:q1" }
  { p: "ex:p1", q: "ex:q2" }
  { p: "ex:p3", q: "ex:q3" }
- Join entry 3:
  { q: "ex:q4", o: "ex:o1" }
  { q: "ex:q5", o: "ex:o2" }
  { q: "ex:q6", o: "ex:o3" }

In the provided example, no triples are obtained when joining entries 2 and 3, indicating that joining entry 1 is unnecessary. Consequently, joining entries 1 and 2 as the first step would result in wasted effort. By carefully selecting the order in which join entries are joined, we can reduce the generation of intermediate results and avoid unnecessary work, which leads to faster query execution.

4

Related Work

The objective of this thesis is to address the performance limitations of Link-Traversal-based Query Processing [11]. We extend this approach with a method of adaptively changing the query plan in order to incorporate approximate join entry cardinalities into the query process. With this approach we aim to improve the efficiency of the query execution process. While we focus our analysis and evaluation on the Solid ecosystem, the concepts and techniques developed in this work may have broader applicability to other decentralization initiatives.

To provide an understanding of the research landscape, it is important to explore existing work and related research in this field. By examining the current state of research, we can identify gaps and opportunities for further advancements. Moreover, it is crucial to outline the distinctive aspects of my work and clarify why direct comparisons with existing approaches may not always be appropriate. By highlighting these differences, we can establish the unique contributions of this thesis.

4.1 Link-Traversal-based Query Processing

Link-Traversal-based Query Processing (LTQP) [11] is a querying approach where Linked Data documents are queried over by following links between them. LTQP was introduced as a way to query the Web of Linked Data as a globally distributed dataspace, without the need to first index it in a single location [26]. In LTQP, the sources are not known in advance. Instead, it employs the follow-your-nose principle of Linked Data during query execution, continuously adding new RDF triples to a local dataset while discovering new sources by following links between documents. As noted in [26], this integrated approach includes parallel source discovery and query execution. It differs from traditional approaches, also called two-phase approaches, that require data retrieval and indexing before query execution.

Compared to traditional query processing, integrated approaches like LTQP cannot rely on pre-execution optimization algorithms that require prior dataset statistics. Instead, it uses a zero-knowledge query planning technique [14] to order Triple Patterns based on link traversal-specific heuristics.

LTQP overcomes the limitations of centralized repository approaches by treating the Web of Linked Data itself as a distributed database. This approach allows for immediate query results, ensures up-to-date data, and harnesses the potential of up-to-date data and data sources. Research in this field focuses on optimizing query plans, as traditional pre-execution optimization algorithms are not applicable in this context.

In the field of LTQP, there is ongoing research that specifically focuses on its application within Solid environments, which is relevant to our use case. One notable finding from this work is that the main area of improvement within LTQP is related to the query plan rather than the discovery of relevant documents [26].

4.1.1 Ranking-Based Traversal for Querying Linked Data

In the field of index-based [15] Linked Data Query Processing, a notable paper focuses on prioritizing URIs to speed up the retrieval of query results [16]. This approach aims to minimize response time by estimating which links contain result-relevant data and processing them first.

It is important to highlight the difference between response time and query execution time. While this paper primarily focuses on optimizing response times, which refers to the time required to find a specific number of result elements, our work emphasizes improving query execution times. Query execution time encompasses the time needed to complete the entire query execution process, ensuring that all discovered documents are fully processed and all results have been emitted.

The paper under consideration investigates the impact of join orderings on the amount of intermediate solutions processed in different test webs or datasets. Surprisingly, the study concludes that optimizing the local work, occurring after the data has been downloaded, has no measurable effect on response times. It identifies the number of requests as being the most important factor in query execution time.

4.1.2 Evaluation of Link Traversal Query Execution over Decentralized Environments with Structural Assumptions

The paper presents a comparison between an integrated approach and a two-phase approach in query execution. The two-phase approach acts as a theoretical case which is not feasible in practice. It makes use of an oracle which provides all query-relevant links which allows it to leverage traditional optimize-then-execute [23, 25] query planning. In contrast to the two-phase approach, the integrated approach relies on zero-knowledge query planning [14]. The two-phase approach is on average two times faster compared to the integrated approach.

The superior performance demonstrated by the two-phase approach can be attributed to its ability to perform traditional query planning, leveraging an indexed triple store with planning-relevant information, including cardinality estimates. In contrast, the integrated approach relies on the zero-knowledge query planning technique which employs heuristics to plan the query prior to execution.

The only difference between the integrated and two-phase approach explored in the paper lies in their respective query planning strategies. This makes it evident that the query plan derived by the integrated approach is highly ineffective. The performance comparison between the two approaches demonstrates the potential for significant performance improvements, with the two-phase approach performing twice as good on average. This highlights the need for improved query planning in integrated execution.

To address this need for better query planning, our work aims to introduce adaptivity into the integrated execution, resulting in a query plan that's dynamically adjusted during query execution. This allows for the integration of planning-relevant information discovered during query execution. By incorporating adaptivity, we aim to decrease the performance gap between the two-phase and the integrated query planning approach.

Another difference highlighted in the paper is the significance of cardinality estimates in the two-phase approach, which contribute to its superior performance compared to the integrated approach. This observation serves as a motivation for our research, as we aim to incorporate cardinality estimates into the integrated query execution process.

4.2 Query federation

Query federation is a technique employed in distributed information systems to facilitate querying over multiple heterogeneous data sources. In a query federation scenario, data sources can be

distributed across different locations and different types of interfaces. The primary objective is to provide users with a unified way to retrieve information from distributed sources as if they were part of a single source. Query federation differs from LTQP, which is specifically tailored for querying over interlinked sets of Linked Data documents.

Traditional query federation approaches typically implement a two-phase approach, where data retrieval and indexing are performed before query execution. In contrast, LTQP adopts an integrated approach that includes parallel source discovery and query execution, eliminating the need for separate data retrieval and indexing phases.

Recent work in the field of query federation has focused on addressing the problem of decomposing a SPARQL query into sub-queries that can be executed by existing endpoints. Some approaches let the decision rely on statistics collected from the sources, approaches also exist that simply consider all possible sub-queries and choose the most promising ones. Other methods implement heuristic-based solutions to identify the sub-queries that can be executed by the available sources or endpoints [21].

Although research has been done to explore integrated approaches in query federation [20], most query federation approaches use a two-phase approach. Therefore, traditional optimize-then-execute techniques as they are often found in the optimization of SQL databases are applicable within this field, allowing for more efficient optimization compared to the Linked Data context. Furthermore, LTQP performs source discovery during query execution, while federated approaches assume prior knowledge of the data sources. Consequently, the performance of query federation is not comparable to that of LTQP.

4.3 Adaptive Query Processing

In traditional query processing, the optimize-then-execute paradigm may fall short in scenarios where reliable cardinality estimates are unavailable. The process of cost estimation heavily relies on accurate cardinality estimates of query sub-expressions. Despite various efforts to improve statistics structures and data collection schemes, many real-world settings suffer from either inaccurate or missing statistics [7]. Another issue that traditional query planning has in a Linked Data context is the lack of upfront knowledge about the data sources. This limitation makes it impossible to apply the traditional optimize-then-execute approaches. Alternative approaches are necessary to overcome these limitations.

Adaptive query processing (AQP) techniques have emerged as a solution to address the challenges

posed by missing statistics and unknown data sources. AQP techniques aim to find execution plans and schedules that are well-suited to runtime conditions by leveraging runtime feedback and adjusting the plan or scheduling space [7] during execution. The objective is to achieve better response times and more efficient CPU utilization.

4.4 Comunica

Comunica[10] is an adaptive query engine which performs part of its query planning during query execution. This adaptivity is necessary as Comunica aims to query remote data sources, where knowing all sources in advance and determining the optimal query plan in advance are impossible. Instead, the choices for query planning are made as soon as the relevant information about the sources becomes available [9].

The work presented in this thesis is situated within the context of an adaptive query engine, specifically within the query planner. The query planner plays a crucial role in optimizing query execution. In subsequent sections, we will identify problems and goals within this context and present our contributions. However, before going into more detail about our work, we will discuss relevant related studies.

4.5 SPLENDID: Query federation optimization using statistical data

SPLENDID [12] is a strategy for executing SPARQL queries over distributed data sources, addressing the challenges of missing cooperation between SPARQL endpoints and the absence of data statistics for estimating query execution plan costs. It enables transparent query federation over distributed SPARQL endpoints and leverages statistical information from VOID descriptions to optimize query execution times.

Unlike our approach, which focuses on the retrieval of pre-generated cardinalities, SPLENDID uses dynamic programming for cardinality estimation. Dynamic programming is a technique commonly employed in traditional relational databases for optimizing the order of join entries.

5

Basic Use Case

The objective of this thesis is to address the performance limitations of Link-Traversal-based query processing [11]. We extend this approach with an adaptive method, changing the query plan during execution in order to incorporate approximate Triple Pattern cardinalities into the query process.

In this chapter, we introduce a use case that involves SPARQL queries in a social network context. The decision to utilize this specific use case was influenced by several factors. Firstly, data and query generation tools, as well as server code were readily available in this context, facilitating the creation and manipulation of datasets for experimentation. Additionally, the use case seamlessly integrated with the query engine utilized in this thesis.

The benchmark used in this thesis, SolidBench, [26] builds upon the well-established Social Network Benchmark (SNB) [8, 5] which models a query workload and data structure as one would find in a social network scenario. SolidBench adds a fragmentation layer on top of the SNB in order to enable it to simulate decentralized workloads, where the data is spread across many Linked Data documents. To illustrate the workload, In the appendix, we have provided examples of queries for reference.

While the solutions presented in this work are specifically evaluated using this benchmark, which simulates the context of Solid, it is important to recognize their generalizability to other Linked Data environments. The approach of using approximate Triple Pattern cardinalities for improved query planning may be applied to various Linked Data querying contexts, benefiting query performance in diverse scenarios.

6

Problem Statement

When executing a query over decentralized data using Link-Traversal-based Query Processing (LTQP), the generation of an efficient query plan is crucial. This plan typically involves many join operations that are responsible for connecting data from different sources. These join operations play a significant role within link traversal query execution due to the heavily interlinked nature of the data.

In practice, join orders in LTQP are chosen using a zero-knowledge query planning technique [14]. Due to the inherent lack of upfront information about the data in the context of LTQP, this choice is often not optimal [26]. Results of this naivety include an excessive number of intermediate join results, causing congestion in the pipeline.

All applications utilizing link traversal query engines to query decentralized data suffer from the same problem since this issue arises in the query engine, which functions as part of their backend. Addressing this issue could benefit LTQP as a whole, facilitating the transition to decentralized data.

The goal of this thesis is to investigate the advantages of exposing structural information about the data pod to the query engine to be used during the query plan generation. The research will define a suitable format for representing the structural information and propose an approach for discovering this information during query execution. Additionally, this thesis will evaluate and compare ways of incorporating the structural information into the query processing pipeline.

The objective of this thesis is to address the performance limitations of Link-Traversal-based querying. [11] We extend this approach with an adaptive method, changing the query plan during execu-

tion in order to incorporate approximate cardinality estimates into the query process.

6.1 Approximating Triple Pattern Cardinalities Using Predicate Cardinalities

Join operations occur on a Triple Pattern basis, meaning that each join entry corresponds to a triple pattern and the efficiency of a join operation. The optimal order of join entries depends largely on the cardinalities of the corresponding Triple Patterns.

To optimize the query plan, it would be ideal to have access to the cardinalities of each triple pattern during the stage of sorting join entries. However, generating precise cardinality information for every possible Triple Pattern is impractical due to the large number of potential combinations. Considering the number of subjects, predicates, and objects as s, p, and o respectively, the total number of possible Triple Patterns amounts to s * p * o.

Our work addresses the challenge of managing cardinality information for all possible triple patterns by focusing on approximating Triple Pattern cardinalities. In the solution section of this thesis, we provide detailed explanations of our approach. However, for now, it is enough to emphasize that we rely on predicate cardinalities as an alternative. One aspect of our work is to explore the effectiveness of utilizing predicate cardinalities as a substitute for Triple Pattern cardinalities.

6.2 Research Questions

The main research question of this thesis is

- How much can we improve the efficiency of Link-Traversal-based Query Processing by incorporating adaptivity and Triple Pattern cardinalities into the query planning process?

To answer the main research question, several sub-questions need to be addressed:

1. Where can we place the predicate cardinalities to effectively leverage them within the query execution process?

2. What are the most effective strategies of combining structural information with adaptive querying techniques?

3. How do the proposed adaptive algorithms perform compared to the zero-knowledge query planning technique [14]?

To address the sub-questions effectively, the following requirements need to be fulfilled:

- The first sub-question requires the extension of existing query execution algorithms to allow discovery and utilization of structural information.
- The second sub-question calls for the development of one or more strategies of integrating the available structural information into the query execution process.
- The third sub-question necessitates a comparison between the zero-knowledge approach and at least one approach related to the second sub-question.

By the end of this work, these sub-questions and their corresponding requirements will have been addressed. By doing so we can gain insights into how incorporating Triple Pattern cardinalities into the query process can enhance the efficiency of Link-Traversal-based Query Processing.

6.3 Hypotheses

The following hypotheses have been formulated in relation to the research questions:

1. Storing a file with predicate cardinalities within a user's Solid pod allows for the query engine to discover it.
2. Extending the query planning approach with algorithms that are based on straightforward concepts, restarting the join operation after a timeout, yields an improvement in the overall query execution time of all queries by 10% or more.
3. There are situations where the overhead of restarting join operations outweighs the speedup achieved by using cardinality estimates.
4. Restarting join entries to sort its entries based on Triple Pattern cardinalities counted during execution improves zero-knowledge querying performance by at least 10%.
5. Sorting join entries with pre-generated predicate cardinalities outperforms zero-knowledge query planning by at least 15% on average.
6. Combining approaches of restarting joins and using pre-generated predicate cardinalities yields better query execution time than each approach individually.

These will be confirmed or rejected by the end of this work.

7

Solution

The core objective of this section is to present our solution for extending the Link-Traversal-based query planning approach [11] with an adaptive method that utilizes approximate Triple Pattern cardinalities to optimize join operations. Our proposed approaches incorporate the concept of restarting join operations and approximating Triple Pattern cardinalities with predicate cardinalities.

Central to our investigation is the exploration of different methods to obtain cardinality information. We analyze the process of counting Triple Pattern occurrences as they appear during query execution and examine the discovery of pre-generated predicate cardinalities. By comparing these two separate approaches, we hope to identify the most suitable method for optimizing efficiency and performance of Link-Traversal-based Query Processing.

7.1 Preliminaries

7.1.1 Elaboration on Cardinalities

In this section, we will delve into the concept of cardinalities, a critical aspect of our solution. It is important to note that we deal with multiple types of cardinalities within the context of our work, specifically predicate cardinalities and Triple Pattern (TP) cardinalities. This distinction has already been touched upon in Section 6.1. In this section we elaborate on the difference and contextualize it within this work.

While Triple Patterns and predicates both play essential roles in the RDF data model, they differ significantly in their characteristics. Triple patterns are combinations of values or variables for subject, object, and predicate, imposing a specific structure on triples. In contrast, predicates represent individual relationships between subject and object. As join operations (Section 3.2) primarily

occur on Triple Patterns, knowing the exact cardinality of a Triple Pattern enables the query engine to make accurate predictions regarding intermediate results during join operations. As a result, TP cardinalities are an ideal criterion for sorting join entries and optimizing query performance.

While TP cardinalities are valuable, creating a file containing the cardinality of each possible Triple Pattern is infeasible in practice due to the sheer volume of available combinations in a Solid pod.

To address this challenge, we turn to predicate cardinalities as a substitute. When joining two entries, we work with the cardinality of each entry's predicate as an approximation of the TP cardinality. We investigate whether this approximation is sufficiently accurate to result in performance improvements of query execution over Solid pods.

7.1.2 Counting Triple Pattern Cardinalities

As explained in detail in section 3.3, knowing how many Triple Patterns are associated with each join entry is crucial for efficient join order implementation. However, due to the lack of upfront information inherent to Link-Traversal-based query processing, this information is generally not available. To address this limitation, we implemented a method of counting the occurrences of Triple Patterns as they appear to allow the query engine to make informed join order decisions, thus improving query execution times.

The first advantage of this approach is its independence from pre-generated predicate information. Unlike predicate cardinalities, which rely on the availability of pre-discovered information, the approach of counting Triple Patterns does not require any pre-existing data. This independence enhances the flexibility and adaptability of the query engine, making it more suitable for handling dynamic and evolving datasets in decentralized environments like Solid.

The second advantage of this approach is its striking a middle ground between the pre-generation of Triple Pattern cardinalities and the use of predicate cardinalities. The former approach becomes impractical due to the vast number of possible combinations, causing increased storage and computation power requirements, while the latter approach was theorized to be potentially suboptimal as predicate cardinalities may not as effective predictors of join result size and join complexity. By counting Triple Patterns as they appear, we aim to harness the benefits of both worlds: obtaining good cardinality estimates without incurring excessive processing overhead or requiring significant additional disk space.

The counting Triple Pattern approach also comes with certain downsides. One significant drawback is that the count needs time to warm up. It requires a time period to accumulate enough patterns

being counted before it can be used to form accurate predictions.

This work aims to investigate the effects of the counting approach when paired with a mechanism which restarts join operations after a certain amount of time. We benchmark different timeout values over a number of distinct queries, measuring when results come in and how the overall query execution time is affected. Further discussion can be found in Section [TODO ref].

7.1.3 Predicate Cardinality File Discovery

In this particular section, more details are provided about the cardinality file. We discover and access its contents in order to obtain accurate predicate cardinalities, which serve as a summary over the entire pod. Code to generate this file can be found at https://github.com/surilindur/catalogue. [13]

It's important to note that while we obtain predicate cardinalities from this file, their practical applicability in estimation join entry cardinalities during query execution might still be uncertain. The reason being that the cardinality of a predicate may significantly differ between the pod as a whole and the specific document we are currently accessing.

Below is an example document containing predicate cardinalities of a pod. This document includes counts of the number of triples, distinct subjects, and distinct objects in the pod. It's important to note that these counts are currently not utilized in our implementation. However, they provide valuable insights into the pod's data structure and may be of use in the future.

Additionally, the document lists the number of predicates for which cardinalities are specified. Although not used in our current implementation, this information could be helpful in estimating the effort required to process the document effectively.

We leverage the VoID vocabulary [2] within our predicate cardinality file, as it was specifically designed for expressing metadata about RDF datasets, making it an ideal choice for our purposes.

```
@base <http://localhost:3000/pods/> .
@prefix void: <http://rdfs.org/ns/void#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<pod> rdfs:type void:Dataset .
<pod> void:triples "2936"^^xsd:integer .      # This pod contains 2936 triples
<pod> void:distinctSubjects "332"^^xsd:integer .  # This pod contains 332 distinct subjects
<pod> void:distinctObjects "110"^^xsd:integer .   # This pod contains 110 distinct objects

<pod> void:properties "1"^^xsd:integer .      # This document lists the cardinality of one predicate
```

```
<pod> void:propertyPartition _:df_0_36031 .    # Definition a subset of the pod
_:df_0_36031 rdfs:type "332"^^xsd:integer .    # In this pod, predicate rdfs:type occurs 332 times
```

To locate the cardinality file, we built upon the *card* file, which contains profile details of the user who owns the Solid pod. We extended this file with a link to the cardinality file. By using such a link, we foster flexibility and customizability, giving each user the freedom to place their cardinality file anywhere, as long as their reference is set up correctly.

For ease of implementation, we decided to create a predicate which is not part of the Solid vocabulary. An example of the link within the *card* file can be found below.

```
@base <http://localhost:3000/pods/> .
@prefix solidt: <http://www.w3.org/ns/solid/terms#> .

<pod/profile/card#me> solidt:voidDescription <pod/profile/voiddescription> .
```

To ensure acquiry of relevant cardinalities without the overhead of accessing multiple indices, we only access the predicate cardinality file of the pod where our search begins, rather than accessing the indices of all pods encountered during the query process. However, it may be worthwhile to look into discovering indices in each pod encountered during query execution. An aspect omitted in our work due to time constraints.

7.2 General Explanation of the Different Configurations in Our Solution

In this section, we present three distinct configurations that aim to improve query execution in decentralized environments.

7.2.1 Configuration 1: Counting Triple Pattern Cardinalities & Timeout

The first configuration introduces a timeout mechanism that starts whenever a join operation is initiated. During the join's execution, Triple Pattern occurrences are counted as they appear, providing the potential for more accurate TP cardinality information to be available once the time limit is reached. The join operation is then restarted, incorporating the newly acquired cardinality information. For these tests, the cardinality sort algorithm is used in both phases.

We have conducted tests using multiple different timeout values, which allows us to investigate the tradeoff between the overhead and benefits of the restart mechanism. For each query, its optimal timeout value was determined empirically, the results of which serve as a theoretically optimal execution time within this approach.

One of the key advantages of the timeout approach, which leverages Triple Pattern cardinalities counted during query execution, is its independence from up-front information provided by the user. This characteristic makes the approach widely applicable and holds promise for its effectiveness in various contexts.

While this configuration offers a valuable comparison between the baseline approach and the potential enhancement achieved through incorporation of counted Triple Pattern cardinalities, it's important to acknowledge that determining the optimal moment for restarting join operations could benefit from further research.

7.2.2 Configuration 2: Predicate Cardinality File

The second configuration, focuses on obtaining predicate cardinalities at the beginning of query execution. In our experiments, each query starts from a Solid pod, which serves as the source for fetching cardinalities. After having successfully retrieved cardinalities, subsequent execution continues as usual, as the newly discovered cardinalities are used to sort future join entries. This approach enables us to assess the impact of having access to the starting pod's cardinalities right from the outset.

The initial implementation of this approach closely resembled the timeout approach, with a callback mechanism to restart the join operation once a cardinality file was discovered. However, it turned out that the cardinality file is discovered at the start of execution, rendering the callback mechanism unused. Despite this, such an implementation does offer the advantage of adaptability and extensibility, especially when combined with functionality to find each encountered pod's predicate cardinality file.

7.2.3 Configuration 3: Combining Predicate Cardinality File and Timeout

The third approach combines predicate cardinalities with TP cardinalities. We start our execution by fetching predicate cardinalities from the Solid pod. During query execution, as Triple Pattern cardinalities are counted, we use them when predicate cardinalities are not available for a given

join entry. Similarly to Configuration 1, we evaluate the ideal timeout to assess the potential benefits of this combined approach.

By exploring these three configurations, we lay the foundation for enhancing query planning and execution in decentralized environments. The subsequent sections will provide further analysis, evaluation, and discussions, showcasing the effectiveness of our solution and outlining potential topics for future work.

8

Results

In this section, we tackle the research question "How much can we improve the efficiency of Link-Traversal-based Query Processing by incorporating adaptivity and Triple Pattern cardinalities into the query planning process?". We simulate Solid data vaults using the benchmark tool discussed in Chapter 5, evaluating different approaches based on the implementation discussed in Section 7. Our experiments take place within the decentralized environment provided by Solid, but findings may be generalizable to other decentralized environments. We begin by introducing the design of our experiment, after which we present our experimental results, followed by a discussion of our results to answer our research question.

8.1 Experimental Design

We conducted an evaluation encompassing six distinct approaches, which includes the baseline approach and a variation of it. The evaluated approaches are as follows:

- **base_zero**: Zero-knowledge based query planning. [14] Also referred to as baseline approach in the remainder of this document.
- **base_card**: Baseline approach, changed to employ a cardinality-based sort algorithm. This approach was compared to the zero-knowledge query planning approach to assess its effectiveness.
- **index_card**: Extension to the baseline, making use of predicate cardinalities as detailed in Section 7.2.2.
- **index_zero**: Index_card approach, instead equipped with zero-knowledge sort. We compare this approach to index_card to able to verify the effectiveness of sorting join entries by predicate cardinalities in isolation.

- **count**: Extension to the baseline which counts Triple Patterns and restarts joins, as detailed in Section 7.2.1.
- **count+index**: count approach combined with index_card approach, as detailed in Section 7.2.3.

In our experimentation, we employed the SolidBench [26] benchmark described in Chapter 5. However, after testing the queries denoted by the benchmark as complex queries, we found that each of them resulted in a timeout. As a result, we made the choice to omit these complex queries from our analysis. We did execute all other queries from the benchmark as part of our investigation.

To evaluate an approach, we executed each query three times and calculated average metrics to obtain reliable results with reduced noise. The measured metrics include timestamps of result arrivals and the total execution time. To prevent excessively long query execution, we set a timeout of 1 minute.

Our experiments were conducted on a 64-bit Pop!_OS 22.04 LTS machine equipped with an 8-core Intel i7-10510U 1.80 GHz CPU and 16 GB of RAM. While the execution of queries over Linked Data on the World Wide Web serves as the primary use case for the concepts discussed in this paper, conducting experiments directly on the World Wide Web is impractical due its unpredictable nature. Therefore, we executed our experiments in a controlled environment. Both the server serving the Solid data and the client performing the queries were executed on the same machine.

8.2 Experimental Results

In our experimental evaluation, we measured the execution times for each query and configuration, as shown in Table 8.1.

	base_zero	base_card	index_zero	index_card	count	count+index
discover-1-0	11072	12410	11170	**2670**	12704	4217
discover-1-1	531	551	547	**484**	792	520
discover-1-2	271	261	289	**241**	411	280
discover-1-3	**9366**	9549	11722	10843	9478	12217
discover-1-4	10268	11278	10526	**3366**	8671	4518
discover-2-0	13630	15230	17880	**3198**	14420	4107
discover-2-1	764	807	750	**483**	877	697
discover-2-2	392	399	425	**322**	414	394
discover-2-3	14467	14595	15038	13997	12515	**9891**
discover-2-4	13660	14558	16395	**4297**	12397	5181
discover-3-0	3	**2**	5	3	**2**	3
discover-3-1	2	**1**	6	60000	2	60000
discover-3-2	5	2	**1**	**1**	**1**	2
discover-3-3	8	10	**5**	60000	**5**	13
discover-3-4	6	8	**4**	**4**	**4**	6
discover-4-0	1	1	1	1	**0**	**0**
discover-4-1	1	1	1	1	**0**	**0**
discover-4-2	1	1	1	1	**0**	**0**
discover-4-3	1	1	1	1	**0**	**0**
discover-4-4	1	1	2	3	**0**	**0**
discover-5-0	3915	3999	3935	3781	**3723**	3992
discover-5-1	216	222	201	**150**	166	213
discover-5-2	116	135	117	**102**	105	113
discover-5-3	3396	3407	3375	**3160**	3240	3239
discover-5-4	3424	3439	3486	**3334**	3359	3401
discover-6-1	423	419	431	143	410	**132**
discover-6-2	260	285	296	111	248	**106**
discover-6-4	11980	12115	12930	8448	11440	**8144**
discover-7-1	445	458	488	190	450	**156**
discover-7-2	293	323	311	150	276	**110**
discover-7-4	18596	18700	20017	15044	9561	**6640**

Table 8.1: Execution time per query of each configuration

	base_zero	base_card	index_zero	index_card	count	count+index
discover-8-0	8355	60000	**7218**	25624	19270	25409
discover-8-1	11468	29084	**8451**	60000	13486	33157
discover-8-2	12945	15130	**6231**	15371	15377	14653
discover-8-3	18704	**9413**	36921	60000	10137	60000
discover-8-4	19160	60000	26702	60000	**15865**	60000
short-1-0	590	603	2081	5630	**10**	17
short-1-1	102	104	1053	2564	**4**	7
short-1-2	85	94	422	2181	**9**	**9**
short-1-3	852	824	60000	6160	**7**	9
short-1-4	438	441	60000	5351	**5**	6
short-2-0	60000	60000	60000	60000	**5081**	60000
short-2-3	60000	60000	60000	60000	**7326**	7525
short-3-0	60000	60000	60000	60000	5	**4**
short-3-1	60000	60000	60000	60000	19	**6**
short-3-2	60000	60000	60000	60000	**8**	34
short-3-3	60000	60000	60000	60000	**6**	11
short-3-4	60000	60000	60000	60000	**28**	262
short-4-0	12	9	9	16	**7**	**7**
short-4-1	6	5	4	7	393	**1**
short-4-2	13	10	9	11	12	**6**
short-4-3	13	**11**	14	18	134	193
short-4-4	1	1	1	1	4	**0**
short-5-0	352	321	415	434	**232**	385
short-5-1	326	305	339	405	**2**	5
short-5-2	676	682	746	782	**207**	428
short-5-3	279	247	314	313	**4**	11
short-5-4	308	**298**	337	372	302	433
short-7-0	60000	60000	60000	60000	26	**12**
short-7-1	60000	60000	60000	60000	12	**10**
short-7-2	60000	60000	60000	60000	357	**24**
short-7-3	60000	60000	60000	60000	**34**	84
short-7-4	60000	60000	60000	60000	**82**	103

Table 8.2: Execution time per query of each configuration (cont.)

The table denotes the best execution time per query in boldface. Queries for which each configuration timed out were omitted for brevity.

Along with this table, we also plotted the execution time in function of different timeout values. An example can be found in Figure 8.1, with more examples in Figures 1, 2, 3 in the appendix.

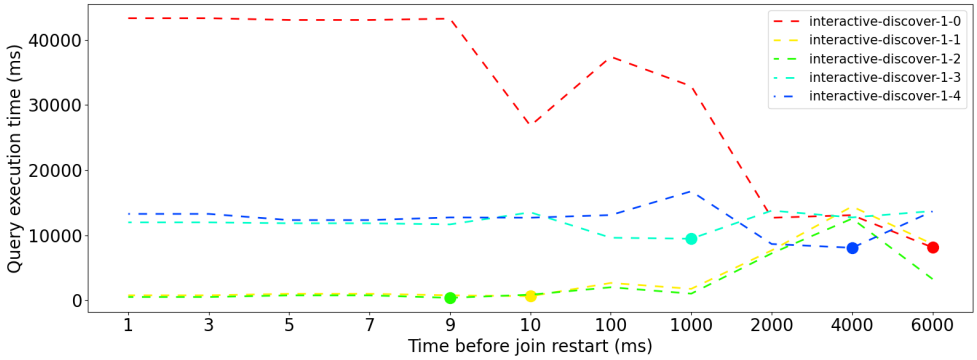


Figure 8.1: Optimal timeout value and execution time for query discover 1 in configuration timeout without index

Table 8.3 presents the comparison of different approaches against the baseline. The first two columns display the percentage of queries for which each approach exhibits execution times at least 10% higher or lower than the baseline, while the last column displays the average improvement in execution time as a fraction of the baseline approach's performance.

	% of queries		
	better	worse	average improvement
index_zero	12.00%	12.00%	-7.86%
index_card	24.00%	16.00%	-91472.15%
count	21.33%	13.33%	-171.86%
count+index	28.00%	12.00%	-73188.57%

Table 8.3: Percentage of queries with execution time of at least 10% better (resp. 10 worse) than the baseline

As we delve into the results, we observe that some queries exhibit distinct patterns of execution which can significantly impact the overall analysis. As such, we decided to base our further analysis of average execution time improvement on the representative subset of queries *discover-1*,

discover-2, *discover-5*, *discover-6*, *discover-7*. We recalculated the metrics from table 8.3, obtaining the results shown in table 8.4.

	% of queries		
	better	worse	average improvement
index_zero	0.00%	5.33%	-6.79%
index_card	20.00%	1.33%	32.57%
count	5.33%	5.33%	0.23%
count+index	14.67%	1.33%	29.39%

Table 8.4: Percentage of queries from representative subset with execution time of at least 10% better (resp. 10 worse) than the baseline

Figure 8.2 shows the amount of queries for which each approach had the best execution time.

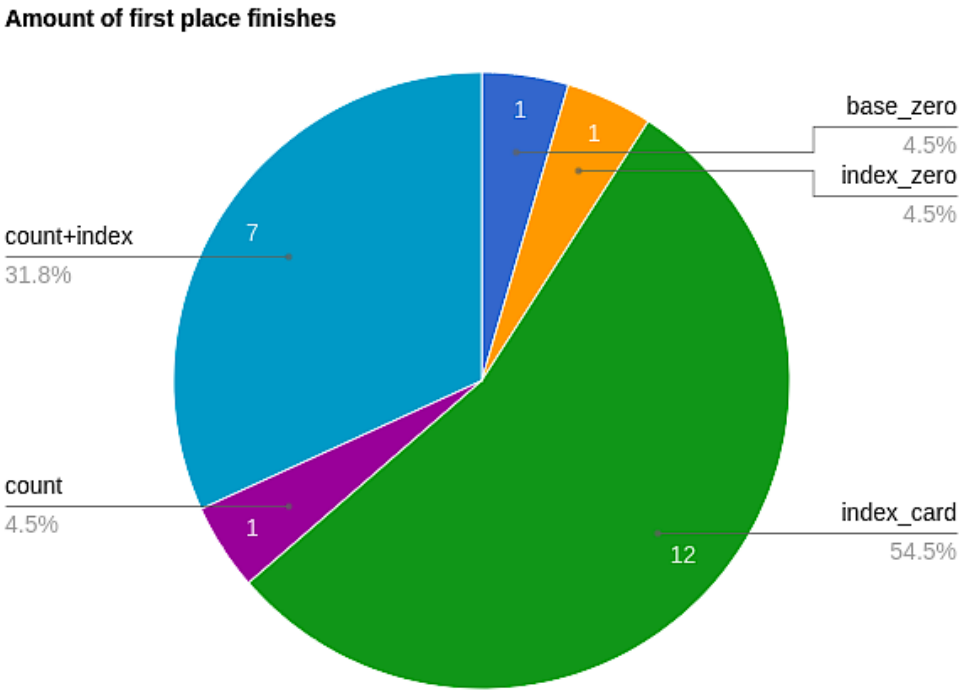


Figure 8.2: Amount of queries where configurations are the fastest

8.2.1 Subsetting the Query Set

We carefully considered the inclusion of each query to ensure the reliability and accuracy of our results. During this process, we identified certain query categories that warranted exclusion from further analysis based on their characteristics. We outline the reasons for excluding these queries below.

Queries with Very Low Execution Time: Some queries exhibited exceptionally low execution times in the baseline configuration, namely *discover-3*, *discover-4* and *short-4*. Due to this phenomenon, even a minor variance in execution time, which might not hold much significance and could potentially be attributed to chance or a minor disturbance, can result in a substantial percentage change in execution time. Including these queries in our calculations severely skewed the averages, as displayed in table 8.5. To maintain the integrity of our findings and provide a more focused assessment, we decided to exclude these queries from further analysis.

	% of queries		
	better	worse	average improvement
index_zero	4.00%	8.00%	-7.70%
index_card	20.00%	8.00%	-104146.55%
count	8.00%	9.33%	-192.84%
count+index	14.67%	8.00%	-83330.31%

Table 8.5: Percentage of queries from representative subset and low execution time with execution time of at least 10% better (resp. 10 worse) than the baseline

Queries Timing Out in the Baseline Approach: As shown in table 8.1, we observed that the baseline approach experienced timeouts for certain query groups, namely *short-2*, *short-3*, and *short-7*. As these timeouts obscure the real execution time and correct amount of results of the baseline approach, we can't compare this to other approaches. To ensure the trustworthiness of our results, we made the decision to omit these queries from further consideration.

Short Queries with Incorrect Results During the course of our experimental evaluation, we encountered a limitation that necessitated the exclusion of certain short queries from the analysis. Specifically, for queries *short-1*, *short-3* and *short-5* an implementation issue related to measuring times resulted in incorrect results for the "count" and "count+index" approaches. As a result, including these short queries in the evaluation would have introduced a significant bias and compromised the integrity of our findings.

Query "discover-8" with Distinct Behavior: During our analysis, an intriguing distinction emerged regarding the behavior of query group "discover-8". This query exhibited higher execution times with the index approach, significantly diverging from the other queries and causing a noteworthy shift in the average execution time as shown in table 8.6. The reason for this difference might be the fact that this query encounters multiple pods during it's execution, it is the only query where this happens and doesn't result in a timeout in every configuration. This is something our implementation was not specifically tailored to. As such, the predicate cardinalities from the starting pod would still be used to guide join order decisions, while the Triple Patterns are coming from the new pod. This effect might be leading to counterintuitive join order decisions and resulting in a heavy increase in execution time. In light of the significant impact of "discover-8" on the overall averages, we intend to exclude it from future evaluations to maintain the integrity and accuracy of our results.

	% of queries		
	better	worse	average improvement
index_zero	4.00%	8.00%	-7.22%
index_card	20.00%	8.00%	-15.33%
count	8.00%	9.33%	-3.82%
count+index	14.67%	8.00%	-8.59%

Table 8.6: Percentage of queries from representative subset and query "discover-8" with execution time of at least 10% better (resp. 10 worse) than the baseline

8.2.2 Discussion

Absence of Globally Optimal Timeout Value

In our counting-based approach, we utilize a timeout value to restart join operations after a certain amount of time has elapsed. In the initial stages of our investigation, our goal was to identify which timeout value would yield optimal query execution performance. However, as we analyzed the results, it became evident that no single timeout value performed exceptionally well across all queries.

Figures 8.1, 1, 2 and 3 illustrate the execution times recorded when using different timeout values. Each figure focuses on 5 variations of a similar query structure, as detailed in Chapter 5. Table 1 shows the complete list of ideal timeout values.

Despite the similarity in query structures, each query exhibits a distinct optimal timeout value. One cause could be the difference in data over which the query is executed.

This outcome emphasizes the absence of a universally optimal timeout value for all queries. In fact, employing a singular timeout value across all join operations within a single query might be unrealistic, given the distinct nature of each join operation.

Given these findings, instead of trying to find the timeout value leading to optimal results for all queries, we determined the optimal timeout value individually for each query. This provided us with a theoretical optimum, to which we then compared the performance of other approaches.

While this configuration offers a valuable insight into the potential enhancement achievable through incorporation of counted Triple Pattern cardinalities, we propose exploring smarter approaches to determine when to restart the query. We anticipate that, by replacing the timeout method with more sophisticated strategies, query execution times can be improved beyond the optimum demonstrated in this thesis.

Interference in Timeout Measurements

As demonstrated in Table 1, a substantial number of queries exhibit high ideal timeout values. In fact, in some of these queries the timeout value surpasses the execution time, which means that in these cases, there is no occurrence of a join being restarted. We can conclude that these particular cases reap no benefit from join operation restarts, which implies that the counted TP cardinalities, acquired during join execution, adversely affect the join order when a restart is triggered.

However, this conclusion might be considerably influenced by the uniform application of the same timeout value to all joins. For instance, a query possessing a join that would ideally benefit from a 10ms timeout might exhibit better execution times when another timeout value is employed, solely because the other join operations do not gain from restarting at the 10ms interval. This facet requires more thorough investigation omitted in this work due to time constraints.

Analysis Within the Representative Subset

The results using the representative subset discussed in Section 8.2.1 clearly demonstrate that the "index_zero" approach is less effective compared to the baseline approach. It shows a 10% performance decline in 5.33% of the queries, while improving none of the queries by more than 10%, resulting in an average performance decrease of 6.79%. These findings align with our assumption

that sorting join entries on estimated cardinality yields more favorable query plans. The evidence supports the importance of considering cardinality information for achieving better query plans.

The "index_card" approach stands out as the most effective among the evaluated techniques. On average, it outperforms the baseline "index_zero" approach by a substantial 32.57%. The results demonstrate the significance of leveraging predicate cardinalities to improve query planning, on the basis of these results we can accept hypothesis 5, saying that this approach performs 15% better or more on average.

The findings indicate that the "count" approach does not consistently lead to significant improvements in query execution time. While it shows over 10% improvement in a small percentage of queries (5.33%), this is negated by it's decrease of 10% or more in 5.33% of the cases and the fact that its overall average improvement is not favorable. This result allows us to reject hypothesis 4 which predicted improved performance. Furthermore, it suggests that counting Triple Pattern cardinalities and restarting join operations to incorporate them alone may not be sufficient for achieving effective adaptive querying.

The integration of the "count" approach with the "index_card" approach, leading to the "count+index" approach, demonstrates an average improvement of 29.39% relative to the baseline approach. The observed decrease in performance allows us to reject hypothesis 6, which suggested that "count+index" would outperform both "count" and "index_card". The inclusion of the "count" approach appears to have an adverse impact on the positive effects offered by the "index_card" approach. This finding implies that the synergy between these approaches is not favorable.

Based on our experimental findings, we cannot accept hypothesis 3 stating that the overhead outweighs the merits of join restarting. While we expect the existence of overhead and expect it to be too small to be significant, the determination of its actual impact remains a topic for future investigation.

In conclusion, the results suggest that the "count" approach is not an effective strategy for adaptive querying, and its combination with the "index_card" approach in the "count+index" approach does not yield promising results. Instead, the findings emphasize that the "index_card" approach, which considers pre-generated predicate cardinalities, is the most effective approach for improving query performance in highly decentralized environments.

9

Conclusion

The primary objective of this thesis was to address the performance limitations of Link-Traversal-based Query Processing (LTQP), caused by a lack of up-front information about the data sources. To achieve this, we implemented different techniques of incorporating cardinalities to improve join order decisions.

We compared diverse methods of acquiring cardinality information in order to identify the most suitable one to enhance the efficiency and performance of Link-Traversal-based Query Processing. This encompassed exploring the process of counting Triple Pattern occurrences during query execution on one hand, and examining the feasibility of utilizing pre-generated predicate cardinalities on the other hand.

We introduced three configurations to optimize query execution in decentralized environments:

The "count" configuration acquires Triple Pattern cardinalities by counting them during query execution, potentially offering more accurate cardinality information over time. Additionally, this approach lets join operations execute for a set amount of time, after which it restarts them, allowing them to employ the counted Triple Pattern cardinalities.

In the "index" configuration, we fetch predicate cardinalities from the starting Solid pod of each query. These cardinalities assist in sorting join entries and optimizing query execution. The approach was implemented with a callback mechanism for increased extensibility.

The "count+index" configuration combines predicate cardinalities with Triple Pattern cardinalities. It starts with fetching predicate cardinalities from the Solid pod, after which it behaves exactly like the "count" method, counting Triple Pattern cardinalities and restarting join operations using a timeout.

To validate our implementations, we simulated Solid data vaults using the benchmark tool dis-

cussed in Chapter 5. We measured execution times of our approaches across a set of test queries. By comparing these results to a baseline approach, we quantified the impact of our adaptive implementations.

Informing join ordering decisions through pre-generated predicate cardinalities obtained from a dedicated file within the Solid pod emerged as the most potent strategy for enhancing query execution times. This affirmation underscores predicate cardinalities' capacity to serve as effective approximations of join entry cardinalities.

The count approach appears to perform similarly to what we would expect from random cardinality assignment. As a result, combining this approach with the "index" approach leads to performance degradation. This outcome implies that solely relying on counting Triple Pattern cardinalities and subsequently restarting join operations to incorporate them does not offer sufficiently accurate cardinalities for guiding better join order decisions.

Our evaluation of the "count" approach across different queries revealed an absence of a universally effective timeout value. Even queries with similar structures didn't share a common optimal timeout. This may be the result of differences in the data over which the queries are evaluated. We conclude that applying a uniform timeout value for all queries, even for all join operations within a query, is infeasible due to the differences between join operations and data specifics. Consequently, we propose researching more intricate methods of determining the join restart timing.

With the pre-generated predicate cardinality files emerging as the most promising solution, we were able to take a step forward toward managing the limitations posed by missing information about the data when querying in heavily decentralized environments.

10

Future Work

In this chapter we discuss limitations of this research and provide avenues for future work.

10.1 Enhancing Restart Strategies for the Count Method

Since our initial exploration of the count approach didn't yield significant improvements, it prompts us to consider the possibility of refining this approach to enhance its results. A promising direction for future investigation involves figuring out more sophisticated strategies to determine the opportune moment for restarting join operations. By moving beyond the timeout-based method into methods which allow a different restart timing for each join, there's potential to surpass the performance demonstrated in this thesis. Discovering methods to improve this approach to a point where it's applicable in real-world scenario's would enable performance improvements in a more general case, without having to rely on the presence of user-generated structural information.

10.2 Refining the Combination of Index and Count Cardinalities

In our current investigation, our focus has primarily been on the pragmatic aspect of combining index and count cardinalities. Our approaches prefer index cardinalities over counted cardinalities whenever they are available, without extensively considering the distinction between the two types of cardinalities.

However, an avenue for future work lies in the possibility of applying more intricate decision-making into the combination. This could involve incorporating counted cardinalities into the approach, provided they have reached a certain level of accuracy during query execution. An important question emerges: at what point in the execution can we deem the counted cardinalities sufficiently

reliable to drive query planning decisions?

This future research direction opens up the possibility of creating a nuanced approach that dynamically adapts its reliance on counted cardinalities based on their accuracy and usefulness. This could lead to even more effective and finely-tuned adaptive query processing strategies, further advancing the realm of decentralized query optimization.

10.3 Cardinality File Discovery in Encountered Pods

Our exploration of index-based methods thus far didn't extend beyond using the starting pod's cardinality file to guide join order decisions. While this approach provided improved performance, future work offers the potential to take indices into consideration from each pod we encounter during query execution, this would lead to an approach applicable in more scenarios.

We look to query *discover-8* as a motivator for future research, as it is the only query that discovers multiple pods during execution and doesn't time out in all configurations. When the "index" approach is used, query *discover-8* performs significantly worse than the baseline approach, distinguishing it from the other queries, which show the opposite trend. We expect that incorporating usage of index files of encountered pods will solve these performance issues, thus making our approach more broadly applicable.

One approach to consider is executing a join restart whenever an index is discovered during query execution. However, this approach may face obstructed query progress when applied to queries that constantly traverse new pods, as the nature of these queries might trigger an exceedingly high number of restarts.

Additionally, the presence of multiple pods could introduce complexities when considering joins with entries drawing from various pods simultaneously. In these cases, the query process will need to recognize that this is taking place and combine the cardinalities from the relevant pods accordingly.

Despite the challenges, research in this direction could broaden the applicability of the optimization

approaches proposed in this thesis.

10.4 Evaluating the Impact of our Approaches On Query Response Times

In this thesis, we evaluated each approach with the main focus on improving query execution times, which is the time to fully process all discovered sources. We neglected query response times, the time to generate a specific number of results, due to time constraints. However, the impact of our research on query response times could be an interesting avenue for future work, as it is a metric that significantly influences the perceived performance of an information system. Improvements in query response times could improve the look of decentralized querying in the public's eye, leading to more widespread adoption.

10.5 Pre-generated Triple Pattern Cardinalities

We have acknowledged that an exhaustive list of Triple Pattern cardinalities is infeasible due to the vast number of possible combinations in Solid pods. However, using the right compression techniques, the size of a file containing Triple Pattern cardinalities might be considerably reduced. This raises the question of whether Triple Pattern cardinalities lead to significantly better join order decisions than predicate cardinalities, and whether this improvement is worth the resulting increased storage usage.

If a lot of disk space is required to save Triple Pattern cardinalities, it may be beneficial to allow users to decide their preferred method individually.

A promising method for reducing required disk space is through the utilization of characteristic sets. [22] The characteristic set of a dataset G and subject s represents the set of predicates p so that a triple s-p-?o exists in G (where ?o is variable). Approaches exist which enable effective prediction of result cardinalities for specific join operations using only the characteristic sets of the dataset. This approach capitalizes on the observation that datasets often utilize multiple predicates to describe the same subject, albeit with different objects, resulting in a relatively small number of characteristic sets in real-world data sets. As a result, incorporating characteristic sets in the future

may allow for reduced storage requirements which facilitate the use of Triple Pattern cardinalities.

10.6 Assessing Approaches against Ground Truth Triple Pattern Cardinalities

An interesting avenue for future exploration involves an in-depth determination of ground truth Triple Pattern cardinalities and optimal join ordering on a per-join basis. Determining what patterns of deviation emerge between the real cardinalities and the approximations could form a comprehensive guide for future research.

This endeavor could involve the creation of an oracle, providing the exact Triple Pattern cardinality values measured for each join during an execution of the query. After collecting the oracle cardinalities, they can be compared with the approximations used in our approaches.

A next step in this comparison could be to evaluate the ordering achieved by our approximations against the theoretically optimal ordering provided by the oracle. The resulting insights could shed light on the extent to which the techniques in this thesis, as well as future work, bridge the gap toward optimal decentralized query execution.

References

[1] Sparql query language for rdf, 2008. URL https://www.w3.org/TR/rdf-sparql-query/.

[2] K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets with the void vocabulary. URL http://www.w3.org/TR/2011/NOTE-void-20110303/.

[3] T. Berners-Lee. Linked data., 2009. URL https://www.w3.org/DesignIssues/LinkedData.html.

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):24–26, 2001. ISSN 00368733, 19467087. URL http://www.jstor.org/stable/26059207.

[5] S. Cheng and O. Hartig. Lingbm: A performance benchmark for approaches to build graphql servers (extended version), 2022.

[6] R. Cyganiak, D. Wood, and M. Lanthaler. Rdf 1.1: Concepts and abstract syntax., 2014. URL http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[7] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007. ISSN 1931-7883. doi: 10.1561/1900000001. URL http://dx.doi.org/10.1561/1900000001.

[8] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742786. URL https://doi.org/10.1145/2723372.2742786.

[9] Ghent University imec. Comunica: Adaptive query planning, . URL https://comunica.dev/docs/modify/advanced/joins/.

[10] Ghent University imec. Comunica: A knowledge graph querying framework, . URL https://comunica.dev/.

[11] Ghent University imec. Comunica: Link traversal, . URL https://comunica.dev/research/link_traversal/.

[12] O. Görlitz and S. Staab. Splendid: Sparql endpoint federation exploiting void descriptions. 2011.

[13] J. Hanski. Github: catalogue, 2023.

[14] O. Hartig. Zero-knowledge query planning for an iterator implementation of link traversal

based query execution. volume 6643, pages 154–169, 05 2011. ISBN 978-3-642-21033-4. doi: 10.1007/978-3-642-21034-1_11.

[15] O. Hartig. An overview on execution strategies for linked data queries. *Datenbank-Spektrum*, 13, 07 2013. doi: 10.1007/s13222-013-0122-1.

[16] O. Hartig and M. T. Özsu. Walking without a map: Ranking-based traversal for querying linked data. 10 2016. doi: 10.1007/978-3-319-46523-4_19.

[17] A. Hogan, C. Gutierrez, M. Cochez, G. de Melo, S. Kirrane, A. Polleres, R. Navigli, A.-C. N. Ngomo, S. M. Rashid, L. Schmelzeisen, S. Staab, E. Blomqvist, C. d'Amato, J. E. L. Gayo, S. Neumaier, A. Rula, J. Sequeda, and A. Zimmermann. Knowledge graphs. synthesis lectures on data, semantics, and knowledge. 2021.

[18] P. Honigman. The social linked data (solid) project of tim berners lee: An organizational take. URL https://hackernoon.com/the-social-linked-data-solid-project-of-tim-berners-lee-an-organizational-take-m94u3z74.

[19] J. Isaak and M. J. Hanna. User data privacy: Facebook, cambridge analytica, and privacy protection. *Computer*, 51(8):56–59, 2018. doi: 10.1109/MC.2018.3191268.

[20] X. Li, Z. Niu, and C. Zhang. Active discovery based query federation over the web of linked data. In Y. Wang and T. Li, editors, *Foundations of Intelligent Systems*, pages 239–248, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-25664-6.

[21] G. Montoya, M.-E. Vidal, and M. Acosta. A heuristic-based approach for planning federated sparql queries. *ISWC 2012 Workshop on Consuming Linked Data*, CEUR-WS.org 2012 CEUR Workshop Proceedings, 2012. URL https://www.researchgate.net/publication/241277898_A_Heuristic-Based_Approach_for_Planning_Federated_SPARQL_Queries.

[22] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*, pages 984–994, 2011. doi: 10.1109/ICDE.2011.5767868.

[23] M. Schmidt and G. Lausen. Foundations of sparql query optimization. pages 4–33, 12 2010.

[24] Solid team. Solid: Your data, your choice. URL https://solidproject.org/.

[25] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, page 595–604, New York, NY, USA, 2008. Associa-

tion for Computing Machinery. ISBN 9781605580852. doi: 10.1145/1367497.1367578. URL https://doi.org/10.1145/1367497.1367578.

[26] R. Taelman and R. Verborgh. Evaluation of link traversal query execution over decentralized environments with structural assumptions. 2023. URL https://arxiv.org/abs/2302.06933.

[27] The Guardian. Cambridge analytica's blueprint for trump victory. URL https://www.theguardian.com/uk-news/2018/mar/23/leaked-cambridge-analyticas-blueprint-for-trump-victory.

Appendices

Appendix A: Execution Time per Timeout Value

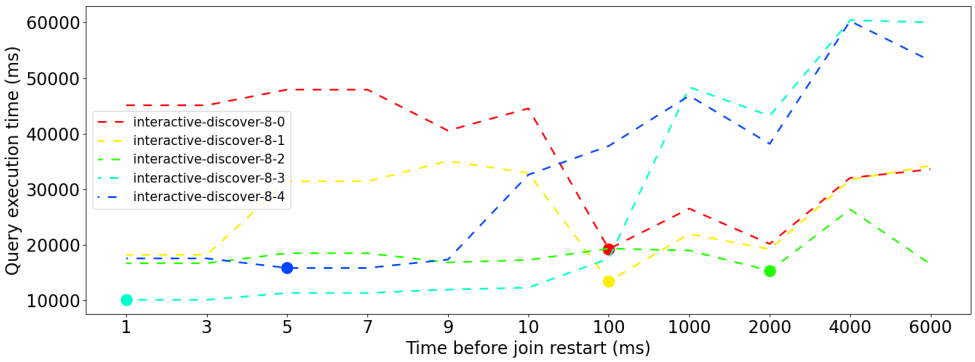


Figure 1: Optimal timeout value and execution time for query discover 8 in configuration timeout without index

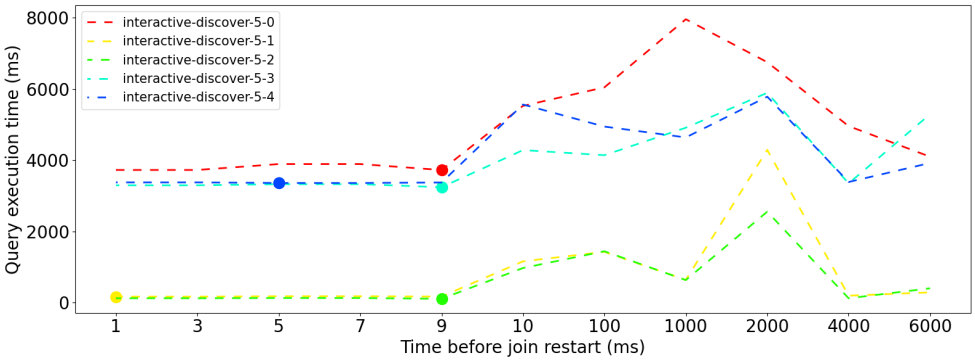


Figure 2: Optimal timeout value and execution time for query discover 5 in configuration timeout without index

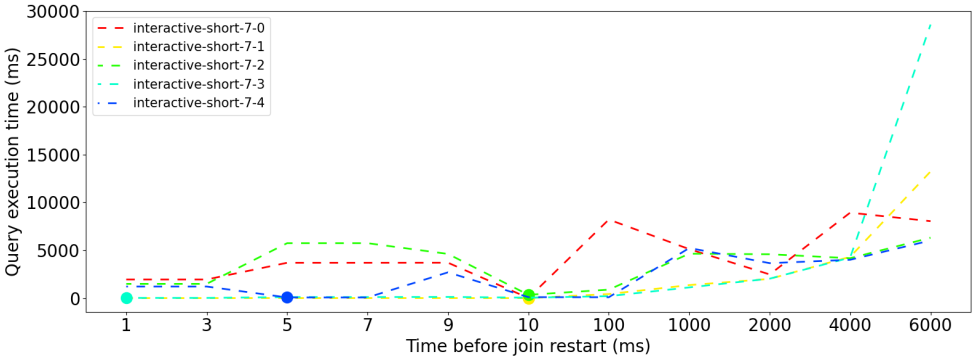


Figure 3: Optimal timeout value and execution time for query short 7 in configuration timeout without index

Appendix B: Optimal Timeout Values

	count	count+index
discover-1-0	2000	1000
discover-1-1	3	6000
discover-1-2	9	6000
discover-1-3	1000	100
discover-1-4	2000	3
discover-2-0	10	9
discover-2-1	9	6000
discover-2-2	9	6000
discover-2-3	1000	100
discover-2-4	10	1000
discover-3-0	6000	4000
discover-3-1	100	6000
discover-3-2	6000	6000
discover-3-3	1000	6000
discover-3-4	100	2000
discover-4-0	6000	10
discover-4-1	6000	2000
discover-4-2	6000	2000
discover-4-3	6000	2000
discover-4-4	6000	2000
discover-5-0	9	6000
discover-5-1	3	6000
discover-5-2	9	4000
discover-5-3	9	4000
discover-5-4	7	6000
discover-6-0	6000	6000
discover-6-1	7	10
discover-6-2	9	10
discover-6-3	6000	6000
discover-6-4	2000	10
discover-7-0	6000	6000
discover-7-1	3	7
discover-7-2	7	10
discover-7-3	6000	6000
discover-7-4	4000	1000

Table 1: Timout value yielding the lowest execution time per query and configuration

discover-8-0	100	10
discover-8-1	100	6000
discover-8-2	2000	9
discover-8-3	3	6000
discover-8-4	7	6000
short-1-0	7	10
short-1-1	3	3
short-1-2	7	7
short-1-3	3	3
short-1-4	3	1
short-2-0	1000	6000
short-2-1	6000	6000
short-2-2	6000	6000
short-2-3	4000	6000
short-2-4	6000	6000
short-3-0	3	7
short-3-1	10	3
short-3-2	3	10
short-3-3	3	7
short-3-4	3	3
short-4-0	6000	1000
short-4-1	4000	6000
short-4-2	10	7
short-4-3	7	2000
short-4-4	10	1000
short-5-0	10	3
short-5-1	7	7
short-5-2	10	1
short-5-3	3	10
short-5-4	7	1
short-6-0	6000	6000
short-6-1	6000	6000
short-6-2	6000	6000
short-6-3	6000	6000
short-6-4	6000	6000
short-7-0	10	10
short-7-1	10	9
short-7-2	10	9
short-7-3	3	9
short-7-4	7	1

Table 2: Timout value yielding the lowest execution time per query and configuration

Appendix C: Queries

Examples to illustrate the queries we evaluated.

All sub-queries of discover-1 have the following structure. The only difference is the card file referenced.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/
    ldbc_socialnet/1.0/vocabulary/>
SELECT ?id ?creationDate ?content WHERE {
  ?message snvoc:hasCreator <POD/profile/card#me>;
  rdf:type snvoc:Post;
  snvoc:content ?content;
  snvoc:creationDate ?creationDate;
  snvoc:id ?id.
}
```

Listing 1: Structure of the discover-1 queries.

All sub-queries of discover-2 have the following structure. The placeholder POD is filled in with a different pod in each case.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/
    ldbc_socialnet/1.0/vocabulary/>
SELECT ?messageId ?messageCreationDate ?messageContent WHERE
    {
  ?message snvoc:hasCreator <POD/profile/card#me>;
  snvoc:content ?messageContent;
  snvoc:creationDate ?messageCreationDate;
  snvoc:id ?messageId.
  { ?message rdf:type snvoc:Post. }
  UNION
  { ?message rdf:type snvoc:Comment. }
}
```

Listing 2: Structure of the discover-2 queries.

All sub-queries of discover-5 have the following structure. The placeholder POD is filled in with a different pod in each case.

```
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/
    ldbc_socialnet/1.0/vocabulary/>
SELECT DISTINCT ?locationIp WHERE {
  ?message snvoc:hasCreator <POD /profile/card#me>;
  snvoc:locationIP ?locationIp.
```

```
}
```

Listing 3: Structure of the discover-5 queries.

All sub-queries of discover-6 have the following structure. The placeholder POD is filled in with a different pod in each case.

```
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/
    ldbc_socialnet/1.0/vocabulary/>
SELECT DISTINCT ?forumId ?forumTitle WHERE {
  ?message snvoc:hasCreator <POD/profile/card#me>.
  ?forum snvoc:containerOf ?message;
  snvoc:id ?forumId;
  snvoc:title ?forumTitle.
}
```

Listing 4: Structure of the discover-6 queries.

All sub-queries of discover-7 have the following structure. The placeholder POD is filled in with a different pod in each case.

```
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/
    ldbc_socialnet/1.0/vocabulary/>
SELECT DISTINCT ?firstName ?lastName WHERE {
  ?message snvoc:hasCreator <POD/profile/card#me>.
  ?forum snvoc:containerOf ?message;
  snvoc:hasModerator ?moderator.
  ?moderator snvoc:firstName ?firstName;
  snvoc:lastName ?lastName.
}
```

Listing 5: Structure of the discover-7 queries.

All sub-queries of discover-8 have the following structure. The placeholder POD is filled in with a different pod in each case.

```
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/
    ldbc_socialnet/1.0/vocabulary/>
SELECT DISTINCT ?creator ?messageContent WHERE {
  <POD/profile/card#me> snvoc:likes _:g_0.
  _:g_0 (snvoc:hasPost|snvoc:hasComment) ?message.
  ?message snvoc:hasCreator ?creator.
  ?otherMessage snvoc:hasCreator ?creator;
  snvoc:content ?messageContent.
}
LIMIT 10
```

Listing 6: Structure of the discover-8 queries.

All sub-queries of short-5 have the following structure. The placeholder COMMENT_URL is filled in with a different pod in each case.

```
PREFIX snvoc: <http://localhost:3000/www.ldbc.eu/
    ldbc_socialnet/1.0/vocabulary/>
SELECT ?personId ?firstName ?lastName WHERE {
  <COMMENT_URL> snvoc:id ?messageId;
  snvoc:hasCreator ?creator.
  ?creator snvoc:id ?personId;
  snvoc:firstName ?firstName;
  snvoc:lastName ?lastName.
}
```

Listing 7: Structure of the short-5 queries.