Semantic Web 1 (0) 1–4
IOS Press

Measuring the potential of client-side adaptive query optimisation for link traversal over decentralised Linked Data documents

Jonni Hanski *, Simon Van Braeckel, Ruben Verborgh and Ruben Taelman
IDLab, Department of Electronics and Information Systems, Ghent University – imec

Abstract. Alongside the emergence of decentralisation initiatives to address issues around regulatory compliance and barriers to entry to data-driven markets, the need arises for client-side query engines, to reduce the overhead of service development atop such decentralised environments, by abstracting away the complexities of data access. These engines, however, are responsible for performant data access in interactive applications, where user-perceived sluggishness can ultimately inhibit the adoption of the underlying decentralisation initiatives themselves. The performance cost consists of the network overhead to acquire the data, and the local processing of it, the latter of which is the focus of our work. Prior work has demonstrated how the structure of certain decentralised environments can assist query engines in efficiently locating and accessing query-relevant data, reducing the relative impact of data access, and exposing the local processing as a major bottleneck. Within this work, we demonstrate the potential of client-side adaptive query planning over decentralised Linked Data documents, using the Solid ecosystem as an example environment. We also consider the impact of request rate limiting and network latency increases, to ensure our findings are also applicable under more realistic circumstances. Through the implementation of a restart-based query planning technique, we achieve average query execution time reductions of up to 15% compared to a baseline of unchanged query plan execution. Through the use of request rate limiting, we also identify optimisation potential in the Comunica query engine framework, with reductions of up to 60% in data transfer and 75% in system resource usage possible through smarter resource allocation. This illustrates the importance and potential of client-side optimisation even in distributed environments, and highlights the importance of further investigation in the direction of adaptive query processing techniques for link traversal.

Keywords: Solid, Linked Data, Link Traversal, Adaptive Query Processing

1. Introduction

With ever-increasing amounts of users, data, and legislation, challenges around centralised approaches to data management are emerging, such as with adherence to privacy-related legislation, or with barriers to entry to data-driven markets, where new entrants would have to collect and manage considerable amounts of data just to provide a competitive service. Together with these challenges, however, a set of alternative, decentralised, and distributed approaches are emerging to address them. One such approach is offered by the Solid initiative [1], where user data would be stored in user-specific online data stores, with the user responsible for managing the data and the access to it. Thus, the development of new and innovative services, as well as further development of existing services, could take place over an open data ecosystem, shifting the responsibilities and costs of data storage away from the service providers, allowing them to focus on the service being provided, instead of the data upon which it is built.

*Corresponding author. E-mail: jonni.hanski@ugent.be.

Decentralisation, however, comes with its own set of challenges. The luxury of uniform and centralised data storage and access may no longer be afforded by developers, adding a layer of complexity previously absent from development of services, potentially leading to additional barriers to entry within the decentralised context. For example, to develop a service that makes use of a specific set of decentralised data, the developers would first need to create or adapt a purpose-built data access framework. One solution to the data access challenge is an abstraction layer in the form of a client-side query engine library [2]. Much like with centralised databases, developers could write declarative queries to manipulate or extract data, and a purpose-built query engine would take on the responsibility of locating and processing the underlying distributed data, ideally without requiring any prior knowledge, or requiring a minimal amount of it. Thus, the engine would be responsible for both the discovery and processing of data, and for this arrangement to be viable in practice, the engine would need to perform its duties with sufficient user-perceived performance. The ideal, zero-knowldge approach to data discovery does, however, limit what can be done optimisation-wise.

Within this work, we investigate the potential of applying optimisations to such a client-side engine within the Solid ecosystem [1], to take one step closer to attaining acceptable performance for practical applications. We evaluate the potential of optimising zero-knowledge client-side techniques, through the use of an example adaptive query planning approach, designed to offer an estimate of the lower bound of optimisation potential. We show how client-side techniques can offer tangible performance improvements, even with more realistic levels of network latency, as well as practical server response rates due to rate limiting. These results demonstrate the need for client-side optimisations, even in decentralised environments with network overhead.

This work is an extension of our earlier investigation [3]. The goal of this extension is to address a combination of observations made during the experiments, as well as recurring reviewer feedback. Specifically, within this extension, we have further optimised the client-side adaptive approach implementation, to ensure lowest possible overhead. Furthermore, we have expanded the set of experiments to address concerns around the relative impact of data access over actual networks, by taking into account realistic levels of network latency, as well as the impact of request rate limiting, in case of server-side rate limits. Our earlier experiments used an experiment setup where the engine was sending HTTP requests to the server at an unrealistic rate, to the extent that would have the client blocked by real remote servers due to request spamming. The earlier experiments also did not take into account proper network latencies, running locally with effectively zero latency. This extension addresses these shortcomings, and underlines the viability of client-side optimisations even in more realistic scenarios.

The remainder of this article is structured as follows. Section 2 briefly discusses related work, followed by Section 3 introducing our research question and hypotheses, as well as Section 4 outlining our approach to tackling them. Section 5 explains our experiments, followed by the results in Section 6. The paper is concluded by a brief discussion and our conclusions in Sections 7 and 8.

2. Related work

Within decentralised environments, where data discovery and processing is delegated to a client-side query engine, the engine takes on the responsibility of carrying out these tasks with sufficient performance for the use case. This section details the related work around data discovery, as well as data processing considerations, and the basics of the Solid ecosystem used as the basis for our work, as an example decentralisation initiative.

2.1. Zero-knowledge data discovery

The zero-knowledge approach to data discovery is enabled by *Link Traversal Query Processing* (LTQP) [4]. This approach bases itself on the *Linked Data principles* [5]. Through widespread adoption of these principles, the World Wide Web enables a globally distributed dataspace in the form of *the Web of Linked Data* [6]. Essentially, when everything is uniquely identified by an IRI, and dereferencing that IRI provides a description of the thing identified by it, the problem of data access becomes a simple IRI lookup task. Link traversal takes this approach, and combines it with link extraction into a so-called *follow-your-nose* approach, where further links to follow are extracted from the data acquired through IRI lookups, using a set of *reachability criteria*. These reachability criteria

include (i) *cNone*, where no links are considered, (ii) *cAll*, where all links are considered, and (iii) *cMatch*, where the links to follow are extracted based on the triple patterns in the query.

With the dataset bounded by the chosen reachability criteria, the query results are only comparable under the specific *reachability semantics* bounded by these criteria [7]. That is, the query results – while not complete within an unbounded Web-wide context – could be considered complete under the reachability semantics used. This is an automated, query-driven approach to browsing networks of interlinked data [8], and is analogous to how a human would browse such a Web of data, yet fully automated and driven by a declarative query language. In practice, this technique allows for resolving queries against Linked Data sources, without prior knowledge of all the data sources contributing to the final answer.

2.2. Zero-knowledge query processing

With link traversal enabling zero-knowledge data discovery and acquisition, executing queries over decentralised data becomes a matter of running the query in tandem with the traversal process, by intertwining triple pattern matching, link extraction, and IRI lookups [9]. Using pipelined query execution, results can be produced as soon as possible, provided the query contains no blocking operations.

Due to the data only becoming available in small chunks during query execution, however, producing an optimised query plan in advance using the traditional *optimise-then-execute* approach to query planning becomes practically impossible. For example, the importance of join planning, as in traditional centralised contexts [10], also applies to decentralised scenarios where the data still has to be processed locally, and excessive numbers of intermediate results affect the overall performance of this processing.

Provided at least some statistics are available during the query planning phase, the *cost and robustness-based query plan optimiser* [11], that seeks to avoid significant performance regressions arising from over-optimistic query planning, could be used to avoid the worst-case scenarios. With absolutely no prior knowledge available, a set of heuristics for *zero-knowledge query planning* [7] may be employed. However, the query plans produced by such heuristics may still differ greatly from the optimum [12].

2.3. Adaptive query processing

Within Linked Data querying contexts, various techniques under the umbrella term of *adaptive query processing* have been successfully employed to address advance planning limitations. The goal of such adaptive techniques is to adapt the initial query plan or its execution to runtime conditions using various forms of execution feedback [13], for example by migrating to a different join order if the data turns out to have characteristics different than expected. The adaptive approaches have been categorised as either *inter-query* adaptivity, for changes between executions, or *intra-query* adaptivity, for changes during execution.

Although inter-query techniques are deemed easier to incorporate into existing *optimise-then-execute* processes, they essentially require executing similar queries over similar data to be able to take advantage of the information acquired. This has been demonstrated through the use of a theoretical oracle in prior work [12], that collects the necessary data prior to formulating the query plan for the actual execution, to achieve up to double the query performance of zero-knowledge query planning.

Intra-query techniques, on the other hand, aim to take advantage of information as it is discovered *during* the execution of a query plan. One such approach is the *postponing of plan selection to runtime*, when the necessary information is available, as described with pre-computed switchable plans [14, 15]. Measurable improvements have likewise been demonstrated through join operation reordering and algorithm replacement [16]. Other approaches include *data partitioning* methods, such as Eddies [17], where different parts of the data are processed using different sets of operators, depending on the data itself. The concept of Eddies has also been applied to Linked Data, such as with the *network of Linked Data Eddies* (nLDE) [18], to process different triples with a different order of operators when possible, in an effort to address fluctuating data access costs over the network, that make cost estimation in advance challenging. Further techniques include *operator-internal* approaches that aim to, for example, allow changing the order of entries in join operations, or changing the physical implementation of a logical join, to reduce the number of intermediate results, such as with the polymorphic bind and hash join operators [11], that

swap between the two strategies at runtime. The Eddies have also been extended with essentially operator-internal techniques [19], to allow for more flexibility in the data-partitioning method itself.

Within this work, we demonstrate the potential of applying intra-query techniques in link traversal over a network of distributed Linked Data documents, using a query plan restart-based approach, to provide a lower bound estimate for the performance potential of adaptive client-side techniques.

2.4. Relative impact of query plan

The overall performance impact of the data processing part has been shown to vary considerably. Some prior work has demonstrated how, in performing traversal-based query execution over Linked Data, the cost of data retrieval over the network marginalises the cost of locally processing that data following the query plan [9]. This conclusion was reached through the application of different static and random query plans on traversal-based query execution over the Berlin SPARQL Benchmark suite data [20], adapted for a number of different test networks with different link structures to them, simulating an online e-commerce environment.

On the other hand, considerable performance improvements have been demonstrated [12] over a social forum dataset, adapted for distributed link traversal scenario from the LDBC social network benchmark datset [21], using a theoretical oracle to produce an optimal query plan with all the required statistics known beforehand. This conclusion was reached in a test Solid environment with negligible network latency, due to the client and server running on the same machine. Nevertheless, the improvements were considerable, with query time reductions of up to half, leading to the conclusion that in specific environments, the network overhead may not fully dwarf the impact of the query plan. Thus, within this work, we focus on the practical performance improvements attainable through client-side optimisations, with the expectation that the overall performance impact will land somewhere between negligible and significant.

2.5. The Solid initiative

The Solid initiative [1] seeks to offer individuals greater control over their own data, by storing it in permissioned personal online datastores, referred to as *pods*, encouraging and facilitating the reuse of personal data, while also enabling users themselves to control access to it. Notably, the Solid pods expose their contents following a set of specifications such as the Solid protocol [22]. The contents of pods are exposed as a tree-like linking structure consisting of containers and resources, using the Linked Data Protocol (LDP) [23]. Additionally, optional, purpose-built Solid type indexes[1] can be used to assist in data discovery. Further work on developing a core Web storage protocol is continuing in the new Linked Web Storage W3C Working Group [24], which may eventually result in data sources outside the Solid ecosystem sharing some aspects of its convenient means of exposing data for machine processing.

This constrained and well-defined means of publishing data in Solid pods has been shown to increase the relative impact of query planning [12], when the environment has enough structure to it that can be taken advantage of to efficiently and quickly locate query-relevant data, shifting the bottleneck from data access to local processing of that data to produce the query results. Further reductions in network overhead have been demonstrated through the use of link prioritisation during traversal [25]. With these considerations, we have chosen the Solid initiative as the basis for our experiments, and the SolidBench benchmark for the evaluation to align with existing work in the link traversal space.

2.6. Triple pattern cardinality estimation

Cardinality information on triple patterns and other algebra operations is used during query planning to determine the join plan – the order of joins – between them, in an effort to minimise the number of intermediate results, thereby also minimising the amount of work needed to process the data. Prior work on the impact of cardinality estimation [10] has shown that, in query plans with multiple joins, errors in this estimation can cause exponential

[1] https://github.com/solid/solid/blob/main/proposals/data-discovery.md

increases in the number of intermediate results. Although centralised storage solutions are often capable of pre-computing such information or providing estimates efficiently, such as through the use of *characteristics sets* [26], within decentralised scenarios this may not always be possible, yet the importance of accurate estimates remains high [12].

Thus, various purpose-built estimation techniques have to be applied in this context, such as *variable counting* [27], that estimates the relative selectivities of triple patterns using the type and number of unbound components, assuming different selectivities for different components of a triple pattern. Other approaches, such as the set of formulae by Hagedorn et al. [28], copied in Table 1, make use of the statistics offered in dataset descriptions published using the Vocabulary of Interlinked Datasets (VoID) [29], in combination with the triple patterns in a given query, to provide more robust estimates in cases where the VoID statistics, such as the number of triples with a given predicate, are available.

Triple pattern	Result cardinality
?s ?p ?o	c_t
subjA ?p ?o	$\frac{c_t}{c_s}$
?s predA ?o	$cp_{predA,t}$
?s ?p objA	$\frac{c_t}{c_o}$
subjA predA ?o	$\frac{cp_{predA,t}}{cp_{predA,s}}$
subjA ?p objA	$\frac{c_t}{c_s \cdot c_o}$
?s predA objA	$\frac{cp_{predA,t}}{cp_{predA,o}}$
subjA predA objA	$\frac{cp_{predA,t}}{cp_{predA,s} \cdot cp_{predA,o}}$
?s rdf:type ?o	$cp_{rdf:type,t}$
subjA rdf:type ?o	$\frac{cp_{rdf:type,t}}{cp_{rdf:type,s}}$
?s rdf:type objA	$cc_{objA,e}$
subjA rdf:type objA	$\frac{cp_{rdf:type,t}}{cp_{rdf:type,s} \cdot cp_{rdf:type,o}}$
or 0 if no class partition for *objA*	

Table 1

Triple pattern cardinality estimation formulae from Hagedorn et al. [28], using `void:triples` (c_t), `void:distinctSubjects` (c_s), `void:distinctObjects` (c_o), as well as their property partition equivalents ($cp_{p,t}$, $cp_{p,s}$, $cp_{p,o}$) for property p, and the class partition `void:entities` ($cc_{c,e}$) for class c.

Within this work, to estimate triple pattern cardinalities for use in evaluating the chosen query plan, and in selecting a new one if deemed relevant, we have chosen to employ a variable counting-based approach due to its lack of preconditions, as well as the formulae from Hagedorn et al. [28] due to their suitability for decentralised scenarios where VoID descriptions can be used to communicate data statistics from a remote server to the client-side query engine. Furthermore, to limit the scope of this work, we are using *single-point* estimates, disregarding any trends over time introduced by unintended correlations and the like, but acknowledge the importance of taking variance – such as the best and worst values for different parameters – into account to produce more robust query plans [15].

3. Research question

Within this work, we seek to explore the impact of applying client-side adaptive query processing techniques in traversal-based query execution over a traversal-friendly decentralised environment. We use a restart-based approach for this purpose, evaluating the current query plan and restarting it if the plan would differ based on information available during the evaluation. The following research questions serve as the basis for our work:

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

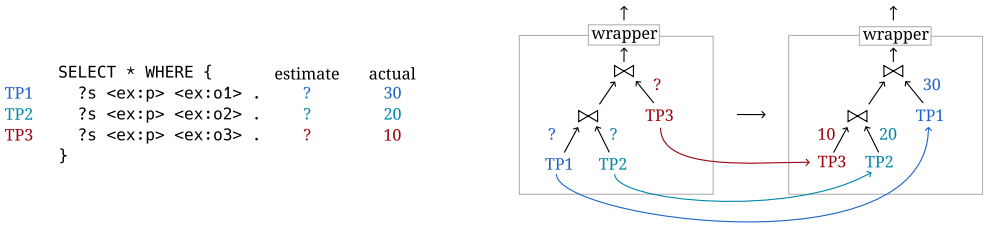

Fig. 1. Once the cardinalities are discovered, and the current plan is suboptimal due to TP1 producing the most results, the wrapper transparently restarts the query plan to place it higher up in the join tree, thus avoiding excessive intermediate results.

Question 2. *How does increasing network latency or limiting request rate affect the impact of these client-side techniques on performance?*

We derived the following hypotheses to answer this research question:

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

Hypothesis 2. *Using a uniform interval for plan evaluation and restart performs worse than evaluation and restart upon the discovery of new dataset characteristics information.*

Hypothesis 3. *Performing plan evaluation and optional restart more than once per query execution will negate any performance benefits due to the associated overhead.*

Hypothesis 4. *Using VoID description metadata, the cardinality estimation formulae from [28] will outperform simple predicate count-based estimation technique, highlighting the importance of the cardinality estimation in achieving performance improvements.*

Hypothesis 5. *Lowering the request rate of the query engine to simulate real-world rate limiting scenarions will marginalise the impact of local optimisations.*

Hypothesis 6. *Uniform increases to network latency will lower the performance benefits of of local optimisations and ultimately negate them.*

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

4. Client-side optimisation and simulation

Within this section, we outline our approach to restart-based query planning, as well as the query plan evaluation approach used, the triple pattern cardinality estimation techniques, as well as the network latency simulation and request rate limiting approaches.

4.1. Restart-based query planning

Within this work, we employ an operator-internal technique to restart query plans from the beginning during *pipelined* query execution, where bindings pass through the query plan one by one, as they are produced and consumed by the operators. Our restart wrapper operator essentially encapsulates the entire query plan, by acting as a virtual passthrough join operator at the top of the *tree of joins* within the query plan, with this tree join of joins consisting of the actual *physical join operators* that form the body of the query plan. This allows the operator to monitor the query plan output, and to transparently restart it without disrupting the pipelines execution at the engine level. This wrapper, illustrated in Fig. 1, is responsible for (i) evaluating the currently executing set of joins to determine whether it is optimal and should be restarted or not, and (ii) restarting the encapsulated query plan when the current one no longer appears optimal.

Our restart-based approach differs from stop-and-restart approaches, such as by [30], in that the query plan is terminated entirely under the wrapper, and restarted from the beginning, with the sole goal of producing a better query plan after the restart. Thus, the approach does not react to resource pressure in shared environments, or allow pausing the query execution. Our work is thereby more analogous to that around Web agents [31], where the agent attempts to achieve higher performance by re-sending a query, in our case by re-running the query with a different plan.

The query plan wrapper operates under bag semantics as required by the SPARQL specification [32], as part of pipelined query execution, under the assumption that a query plan, when restarted, produces its full output again from the beginning. The wrapper internally keeps track of all output produced by the query plan it encapsulates, using a mapping of bindings to their produced counts. Upon restarting the query plan, the wrapper uses this mapping to discard bindings that were already produced by the previous execution, ensuring no spurious duplicates are produced, without dropping intended future duplicates. While this record is maintained fully in memory within our implementation, and thus is unsuitable for use with queries producing more bindings than can be stored in system memory, practical solutions could look towards flushing it to disk as with XJoin [33] or agjoin [34], or splitting the record across several memory pools based on hashes such as with GRACE [35].

4.2. Query plan evaluation

Alongside restarting the query plan, the wrapper operator is also responsible for evaluating the optimality of the chosen query plan. This is achieved by creating a new query plan at the time of evaluation, given the triple pattern cardinalities available at that moment, that the engine deems optimal. This current optimal plan is then compared against the executing plan, and if they differ – for example, if the join order between two triple patterns is different – the wrapper considers the current plan sub-optimal. Should the plans be identical, the current one is still optimal, and can continue execution.

The wrapper can be configured to perform its query plan evaluation using two different approaches, based on our hypotheses:

1. *Interval-based*: When the query plan is initially started, the wrapper sets a recurring timeout at specific intervals. Every time the timeout is reached, the query plan evaluation takes place. For example, if the interval is set to 100 milliseconds, the wrapper will evaluate the current plan after 100 milliseconds. If the plan still remains optimal, the wrapper will wait an additional 100 milliseconds, and perform the evaluation again after 200 milliseconds of total execution time.
2. *Update-based*: Every time the cardinality estimate of a triple pattern is updated, such as when the query engine discovers a VoID dataset description applicable to the currently executing query, the query plan evaluation is carried out. This approach allows for the evaluation of the query plan only when the information affecting query planning is updated, and should ideally perform fewer unnecessary evaluations than the fixed interval approach.

Both approaches can be configured to carry out the evaluation an unlimited number of times, or only a specific number of times. Additionally, the wrapper performs cardinality estimation for the entire query plan it encapsulates, when the cardinality of the individual join entries is updated, and uses this total cardinality estimate as the basis to evaluate *how much work has already been done* by the query plan. When the number of produced bindings relative to the total estimate for the query plan is above a configurable percentage threshold, the wrapper will skip restarting the query plan, to avoid scenarios where the query plan would be restarted when almost all of the work has already been done.

4.3. Triple pattern cardinality estimation

The query plan evaluation relies on cardinality information on triple patterns being updated as new information becomes available. If the cardinality information does not change, the evaluation will produce the same plan every time, and the experiments would be identical. Ideally, any new cardinality estimate would be closer to the true cardinality value known only at the end of the processing.

Within this work, we have chosen to employ the following two triple pattern cardinality estimation techniques, taking advantage of the information provided by VoID dataset descriptions:

1. A *formula-based approach*, that uses the the formulae from Hagedorn et al. [28], included in Table 1. The edge cases not covered by the formulae, or cases with missing statistics, or where the divisor in a formula would go to zero, the total number of triples in the dataset is used as the realistic, conservative upper bound estimate.
2. A *predicate-based approach*, that assumes the cardinality of a triple pattern to equal the cardinality of the predicate value within that triple pattern. For example, if the triple pattern has a predicate `ex:p`, and the VoID description contains a predicate partition for `ex:p` with triple count *n*, then *n* is used as the cardinality estimate for this triple pattern. For triple patterns with variable predicate, the total number of triples in the dataset is used as an upper bound estimate.

These two approaches will allow us to establish an understanding of the impact of cardinality estimation techniques on restart-based query planning, to avoid accidentally evaluating the cardinality estimation approach rather than the restart itself. The initial query plan with both cardinality estimation approaches, prior to execution, is produced based on zero cardinalities for all triple patterns and query operations, Whenever new information becomes available, such as when a VoID dataset description is discovered, the cardinality estimate is updated immediately. The cardinality estimator also breaks down larger chunks of the query into triple pattern level if necessary, and reconstructs the higher-level cardinality as a worst-case estimate of the lower-level components. This allows the engine to perform cardinality estimations on, for example, a union, by estimating the total cardinality as the sum of the input cardinalities.

4.4. Network overhead simulation

Within this work, we employ two types of network overhead simulation: (i) request rate limiting, to simulate a more realistic client-server interaction, and (ii) network latency simulation, to emulate realistic levels of delay between a request and a response. These will allow us to better evaluate the scaling of client-side optimisations under more realistic scenarios with regards to the data access overhead over the network.

The request rate limiter operates by matching client request rate to server response times: if the server responds to ten requests a second on average, the rate limiter will space out client requests to reach a corresponding average of ten requests per second. The rate limiter keeps track of the server response time, using a simple smoothing multiplier to shift this calculated response time towards the latest measured value. This uses the method in (1) to calculate the interval applied between latest request n and the next request $n+1$, where S is the constant smoothing multiplier.

$$i_{n+1} = \begin{cases} 0 & n = 0 \\ i_0 & n = 1\ , S \in]0,1] \\ i_{n-1} + S \cdot (i_n - i_{n-1}) & n > 1 \end{cases} \tag{1}$$

This allows for dynamic rate limiting without relying on server-side communications, and results in the query engine exhibiting more polite client behaviour, as opposed to sending as many requests as possible to a server, potentially overloading smaller servers or taking resources away from other clients.

The network latency simulator component is independent from the request rate limiter, and applies a uniform, configurable delay to all requests. For example, if the latency simulator is configured for 10 milliseconds, then each outgoing request will be delayed by 10 milliseconds, in addition to any rate limits or normal underlying network latency. This allows for the simulation of realistic levels of network-induced data access slowness.

5. Experiment setup

Within this section, we describe the benchmark used, our implementation of the approaches, as well as the experiments conducted.

5.1. The SolidBench benchmark and VoID descriptions

The dataset and queries used for our experiments were generated using SolidBench[2], a benchmark to simulate a distributed social network use case across Solid pods using the LDBC SNB social network dataset from [21]. This benchmark has been used in related work [12], as well, for evaluating the relative impact of query optimisation, thus proving suitable for our use case.

The generated dataset consisted of 1,528 Solid pods, containing a total of 3,514,190 triples across 117,967 documents, with an average of 30 triples per document and 77 documents per pod, excluding the LDP container structure links that were generated on-the-fly by the Community Solid Server [36] used to serve the dataset. The SolidBench benchmark also uses a set of 27 query templates – 7 short queries, 12 complex queries and 8 discover queries – to instantiate 5 queries per template for a total of 135 queries, based on the generated data.

Following the work by Hagedorn et al. [28], to enable the use of the cardinality estimation approaches detailed earlier, we chose to expose dataset metadata using the VoID vocabulary [29] for each Solid pod within the benchmarking dataset. We extended the RDF dataset fragmenter library used by the SolidBench dataset[3] to also generate VoID descriptions, and treated each pod as a dataset. These descriptions were placed at the pod roots as metadata, and thus served by the Community Solid Server when a client requests the pod root URI. This enables the automatic discovery of these descriptions during traversal over the pods.

5.2. Query engine implementations

The engine-level approaches discussed in Section 4 were implemented in Comunica [37], a modular SPARQL query engine framework that provides a baseline link traversal implementation, also previously used to benchmark the relative impact of query plans compared to network overhead in related work [12]. The Comunica framework allowed us to implement only the components needed to evaluate our specific approaches, ensuring a fair comparison between the different scenarios from a technical standpoint. Our implementation is available as open source software[4].

Through changes in the query engine configuration, we set up the following test cases to measure the impact of our approaches:

- The *baseline* approach, using the heuristic-based zero-knowledge query planning approach [7]. This is the standard configuration of the engine without any of our implementation overheard, and performs link traversal under the *cMatch* reachability criteria, where links to follow are chosen based on triple patterns in the query.
- The *overhead* evaluation, otherwise identical to the baseline setup, except with VoID description parsing and cardinality estimation using either the predicate count-based approach or the formulae from Hagedorn et al. [28] The zero-knowledge heuristics for query planning are also replaced by a cardinality-based approach, but with all cardinalities being zero during the initial planning phase. This is a configuration with all the implementation overhead necessary for cardinality estimation and updates, but without taking advantage of it to evaluate the query plan or to restart it. This allows us to measure the impact of our implementation itself, to ensure it does not unnecessarily skew the results in either direction.
- The *interval-based restart* approach, identical to the overhead experiment in its configuration, but with the evaluation and potential restart of the query plan taking place at specified intervals. This experiment is conducted for single-restart and unlimited restart scenarios, to measure the overhead of multiple restarts.
- The *update-based restart* approach, also identical to the overhead experiment in its configuration, but with query plan evaluation and potential restart taking place whenever the cardinality estimate for a triple pattern is updated. This experiment is also conducted for single-restart and unlimited restart scenarios, to measure the overhead of multiple restarts.

[2] https://github.com/SolidBench/SolidBench.js/tree/3e45198
[3] https://github.com/SolidBench/rdf-dataset-fragmenter.js/tree/9b3f2ae
[4] https://github.com/surilindur/comunica-components/tree/d6e936a

5.3. Experiment overview

The following parameters were varied between the experiments, to get a more complete overview:

- The join ordering approach for creating query plans, varied between *heuristics* [7] and a simple *cardinality*-based ordering modes.
- The cardinality estimator, varied between no estimation using heuristics, the *formulae* from Hagedorn et al. [28], and the *predicate*-based estimator.
- The query plan evaluation scheduling, varied between *interval*-based approach at 100, 1,000 and 10,000 millisecond intervals, and the estimate *update*-triggered approaches.
- The restart limit, varied between unlimited restarts a one-time restart for the interval-based approaches, and an unlimited restart count for all estimate update-based approaches.
- The additional network latency, varied between no added latency, 50 milliseconds of latency, and 100 milliseconds.
- The client-side rate limiting, varied between enabled and disabled.

These variations are detailed in Table 2 for clarity. The threshold of estimated work done, after which join restart would not be allowed anymore, was set to 33%, as a balance based on manual testing, that allows for effectively all join restarts to take place, save for those that would clearly be done unnecessarily late in the execution. The choices of additional network latencies were based on observations of real-life latencies, with 50 milliseconds corresponding to latency between different parts of a continent, and 100 milliseconds corresponding to latencies between different continents or just latencies over a slower network within the same continent. We believe these values offers a sufficient estimate of values encountered in real world use cases.

5.4. Experiment environment

The experiments were all executed on the same virtual machine, with a 4-core 8-thread AMD Milan EPYC™ 7003 CPU, 32 GB of DDR4 memory, and an NVMe SSD for data storage, running Fedora Server 42 on linux 6.14.11 kernel with security mitigations enabled. Both the Community Solid Server and Comunica query engine client were running locally and communicating over a Docker network, using the *jbr.js*[5] benchmark runner tool [12].

Query timeout was set to 60 seconds, which is reasonable for the nature of the benchmark: the goal is to simulate a social network scenario, where a user issues queries to discover messages, comments, reactions or other data from this social network, and it is reasonable to expect these search-like actions to complete within one minute, as the low user-perceived performance would otherwise render the social network application unusable. Although this is stricter than the 120-second timeout from related work [12], from a usability perspective in an interactive application, having to wait for over a minute for a search to finish could be deemed unusable. The queries instantiated using the complex query templates timed out even with 120-second execution limit, aligning with the prior work, leaving a total of 75 short and discover queries to execute for each experiment configuration. The experiments and our results are available online[6] for reproducibility and validation.

6. Results

The results overview is available in Table 3, with the full results available online alongside the experiments. Different experiment test cases successfully completed different sets of queries within the allocated budget of time and system resources, and thus the analysis has been limited to the queries successfully executed by all test cases. Even the baseline configuration was unable to complete all queries. Of the total 75 queries, a common set of 16 queries was thereby left for analysis. We will analyse these results from the perspective of execution time, result production efficiency over the execution, system resource consumption, and network usage.

[5] https://github.com/rubensworks/jbr.js/tree/de543aa
[6] https://github.com/surilindur/comunica-experiments/tree/5c7539a

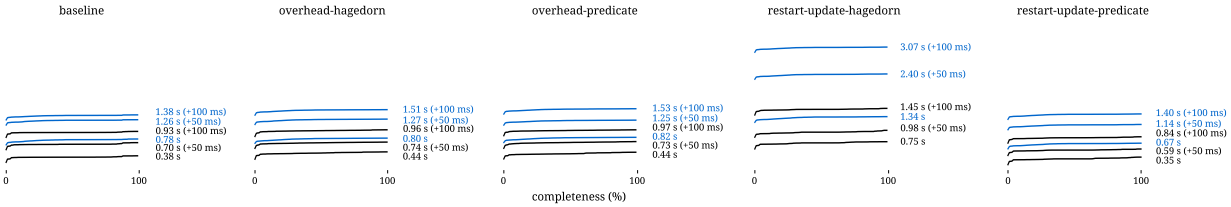

Fig. 2. The average completeness curves for different experiments, with client-side rate limiting and without it. The experiments with higher added network latency exhibited greater variance of the average, as well, although the averages themselves are relatively close to each other. One can observe how the shapes remain the same as latency increases, meaning latency mostly serves to slow things down, on average, rather than significantly alter query behaviour.

6.1. Query performance

From the results, one can observe how the change from a heuristics-based join ordering to cardinality-based one for the overhead experiments only had a modest impact on average query performance and resource consumption. This impact includes the overhead of our implementation, as well as the reduction in potential query planning performance caused by the use of zero-cardinalities for the query plan. The average execution time increased by mere 11%, with increases in network latency further reducing this to around 2%. The first result was also produced around 12% slower, with the difference reduced to around 3% by additional network latency. The same trend continued with last result, that was produced around 16% slower, reduced to 4% with network latency. The maximum query execution time increased by mere 17...29%, and the maximum time to first and last result by 11...50%. The queries themselves are written in a way that is already quite optimal, so this comes as no surprise, and with identical estimated zero cardinalities, the engine has no means of determining an alternative join order. Still, the heuristics applied in the baseline experiments were able to reduce the worst-case performance by up to one third, which highlights their importance in the absence of cardinality estimates.

The general trend appears to be for the average values for query duration, as well as first and last result, to be higher with the restart-based approach. This can be seen from Fig. 2, where the completeness curves are shifted upwards as a whole across the board. For example, evaluating query plans immediately upon cardinality estimate updates, using the formulaic approach for the estimates, ended up increasing the average execution time by 32...71%, producing first result 44...113% slower, and last result 40...97% slower on average. The minimum values, however, have remained essentially identical to baseline, and the increased average can be attributed to the significantly worse maximum execution times at 2.4× to 4.5× the baseline, as well as the corresponding increases of up to 4.3× in times to first and last result. This is likely caused by overly optimistic estimates using the formulae, that may be more accurate at the time of estimation, but that take into account full triple patterns and thus produce estimates lower than the predicate-based approach. Thus, every time new information becomes available, the optimistic triple pattern cardinality estimates may increase in a non-uniform fashion across the query, and have a higher chance of changing their relative order, thus triggering a restart, increasing the overhead and exacerbating the worst-case performance.

The predicate-based cardinality estimation approach manages to avoid the worst-case regressions, by producing unnecessarily conservative cardinality estimates to the extent that their relative ordering is unlikely to change, unless the underlying data distribution characteristics dramatically change as a whole, which does not happen within the context of our benchmark. Thus, the predicate-bases estimator, upon plan evaluation after cardinality estimate update, produced 10...15% lower execution times on average, depending on network latency, and the first result 11...16% faster, and the last result 8...14% faster on average. While the minimum query duration was effectively identical to baseline heuristics, as were the minimum times to first and last result, the maximum query durations were 31...35% lower, the maximum times to first and last result 12...38% lower. This is a key observation in our work, and could easily be ignored if only looking at the averages. On the other hand, the predicate-based estimator failed the most queries, unless additional network latency or rate limiting was applied, resulting from unnecessarily conservative query plan selection in fear of its own grossly overestimated cardinalities. The queries that worked, however, and thus were included in this analysis, did so much better thanks to this.

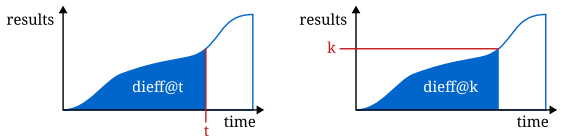

Fig. 3. The *diefficiency metrics* are calculated as an integral of results arrivals over time. For diefficiency at time t, higher is considered better, as the engine has produced more results within that given time. For diefficiency at k results, lower is considered better, as the engine has spent less time producing the given number of results.

Evaluating the join plans at fixed intervals followed a pattern similar to evaluations upon cardinality estimate updates. The formulaic approach to cardinality estimates resulted in significantly worse worst-case performance, lifting the averages up with it. Notably, the performance when restricting restarts to take place only once was almost identical to the performance of unlimited restarts, underling the impact of the initial overly optimistic estimates to the entire execution. With increases in the evaluation interval, the worst-case performance of the forumlaic estimator improved considerably, due to it being able to perform more accurate estimates, catching up to the predicate-based estimator. The predicate-based estimator, on the other hand, performed more consistently, staying relatively close to the baseline performance in its best and worst cases, but also not providing significant improvements. Thus, it can be concluded that an interval-based solution remains suboptimal, as was the case also in our prior work.

Applying client-side rate limiting does not change the overall trend of the results, as seen from Table 3, but rather increases all query durations, and times to first and last results, across the board. The absolute worst-case performance becomes worse, with the outliers shifted further out. This can also be seen from Fig. 4, where the individual query execution times are plotted. The client-side rate limit spreads the execution times away from the lower-end clusters, towards the higher values. Applying uniform network latency increases has a similar impact. Still, even with increases in network latency and decreases in request rates, the trend remains the same, and tangible improvements can be attained through client-side optimisations. With client-side rate limiting applied, and a network latency of 50 milliseconds, query plan evaluations upon cardinality estimates using the predicate-based estimator achieved 8% lower average query execution times, and 22% lower worst-case execution times. The average time to produce the first result was also 11% lower, and the worst-case first result was produced 3% faster.

Moving up to 100 millisecond network delays does negate the impact of client-side optimisations when rate limiting is also applied, but with an average HTTP request count of 62 per query, this would translate to around 6.2 seconds spent on HTTP requests on average, when the average query execution time is around 1.9 seconds and the worst-case execution time around 5...6 seconds. With the rate limiter trying to avoid excessive concurrent request counts, as a server-side rate limiter implementation would do the same, this shifts the majority of the network latency directly into rate limits, thus increasing the relative cost of data access considerably.

Beyond execution times, and times to first and last result, we chose to employ the *diefficiency metrics* [38]. These metrics capture the continuous query execution efficiency, over the duration of the query execution, and allow comparison between query executions that have identical total duration, but that produce the results with different result arrival trends. This helps us gain additional insights into the behaviour of the different test scenarios. The original paper introduced two metrics: 1. *dieff@t* for diefficiency at a point t in time, where higher is better, and 2. *dieff@k* for diefficiency at the time of producing k results, where lower is better. The metrics are calculated as an intergal of the result distribution function over time, illustrated in Fig. 3.

Within this work, we chose to employ *dief@k*, and assign k to the total number of results for a given query. This allows us to aggregate the results across queries producing different numbers of results, while still maintaining a connection between the metric and the continuous efficiency represented by it. These diefficiency results are plotted for every execution in Fig. 4, to provide the full overview of their distribution, with and without client-side rate limiting applied.

From the results, one can observe how the diefficiency metrics generally follow the trend exhibited by query durations, save for the outliers. For example, for the estimate update-based restart experiment, using the formulaic estimator, the diefficiency values remain lower than without rate limits, even though the execution time increases. This would indicate better continuous efficiency over the query duration, despite the duration itself being longer. For queries producing results in the range of a hundred or so, such as the *interactive-discover-2* template, the results

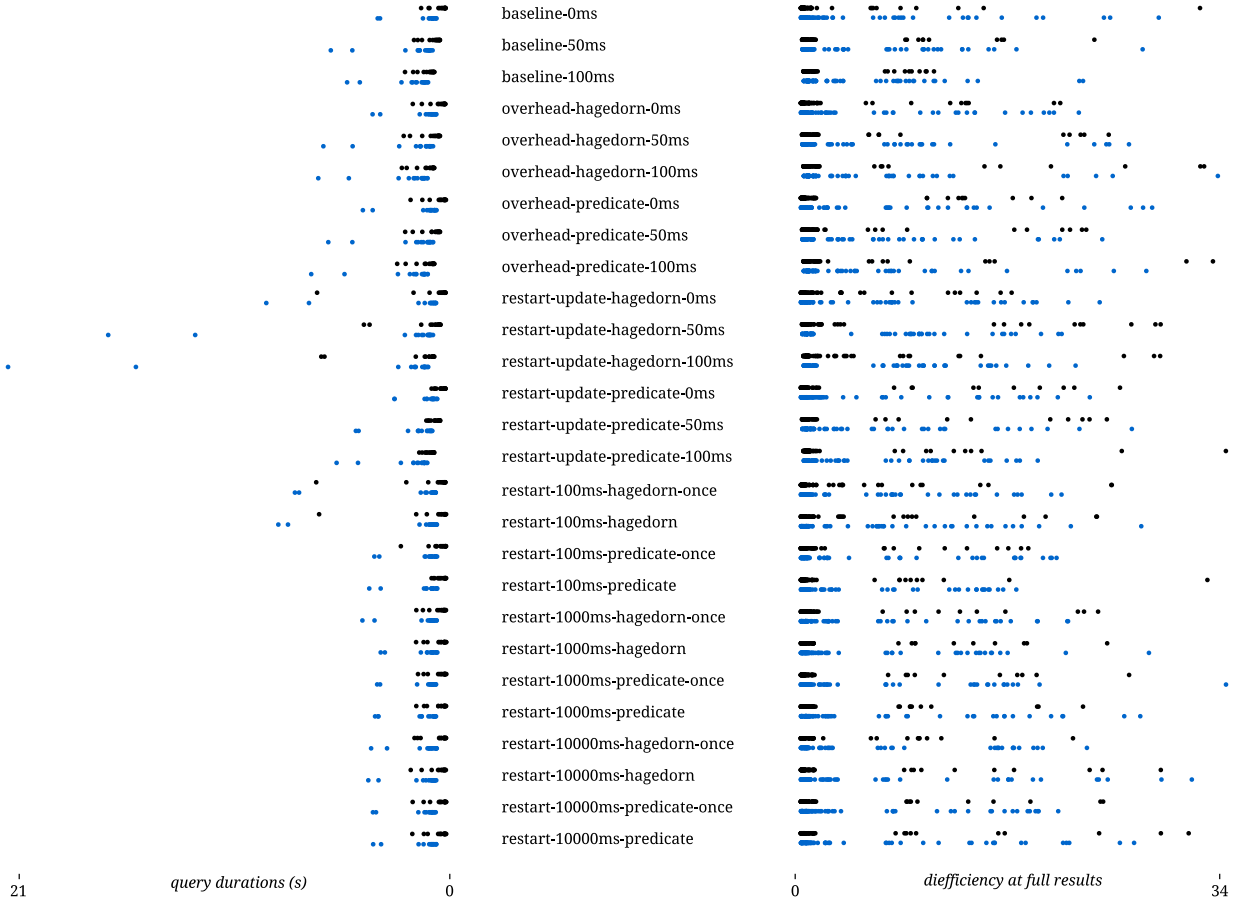

Fig. 4. The individual query execution durations and the corresponding *dieff@full* values, both with client-side rate limiting and without it. Applying rate limiting causes long-running queries with many HTTP requests to take even longer. Furthermore, the diefficiency values are spread more towards the higher end, also due to the increased query duration.

are indeed produced more smoothly over time, despite the query execution itself taking slightly longer. This is illustrated in Fig. 5. From a user perspective, this gradual delivery of results could make for a better experimence, and help the potential application taking advantage of such querying appear more consistent in its performance.

Further analysis of the rate limiter approach, as well as the network latency configurations, suggests that in the absence of a rate limiter, the engine dedicates most of its time to link traversal and document parsing, and focuses fully on query execution only when traversal subsides. This could be caused by the Comunica query engine framework used for the experiments using the Node.js environment, relying on the event loop for mostly single-threaded processing of both the query and the traversal, as well as the document parsing. We believe splitting the traversal and parsing to a separate thread – such as a worker process – should help avoid this infighting for resources, but there was no simple way to test this due to the interconnected design of the engine, and the potential overhead of transferring data between worker threads.

6.2. System resource consumption

The *CPU-seconds* metric captures the consumption of CPU resources over the duration of the experiment, and the *GB-seconds* metric captures the consumption of memory over the same duration. Due to the way the CPU and memory consumption is recorded by *jbr.js*, these metrics could not be connected to individual query executions, and thus capture the entirety of an experiment, including failed queries. Nevertheless, being uniform across the experiments in its behaviour, the values can still be compared, and help understand the hardware costs of local

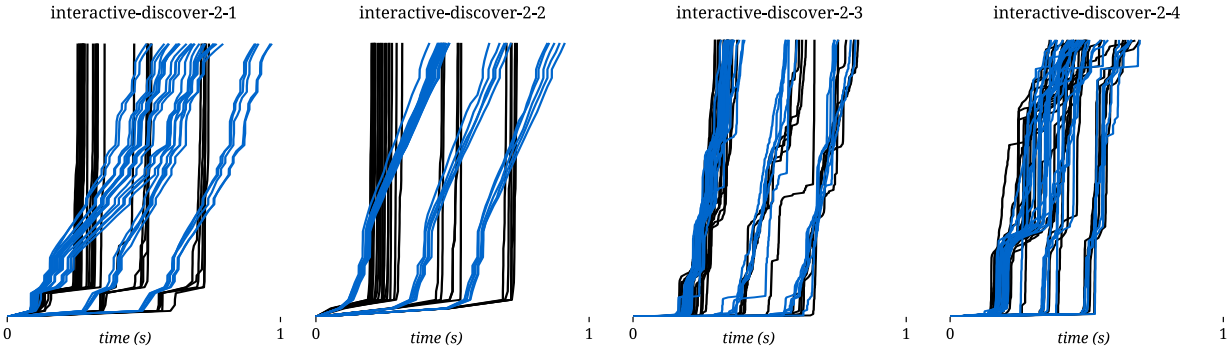

Fig. 5. The result arrival trends for various experiments, both with client-side rate limiting and without it. The rate limiter causes results to be produced more smoothly over time, whereas the default configurations appear stuck at first, and then suddenly dumps the majority of results in one go.

query processing. The metrics also help understand the relative cost of executing queries in environments that bill services by their resource consumption.

From the results in Table 3, one can observe how the overhead experiments exhibited around the same resource consumption as the baseline, achieving an average 10% lower CPU-seconds and 5...30% lower GB-seconds in total across the benchmark runs. This is contrast to the query durations, where the overhead experiments exhibited slightly longer execution times, and thereby indicates differences in query plans that result in lower resource consumption at the expense of longer execution times. Restarting query plans upon cardinality estimate updates resulted in CPU-second increases of 13...36%, whereby the formulaic approach to cardinality estimation mostly increased processor utilisation. The associated memory consumption was likewise increased by 20...36%. This aligns with the poorer query performance overall, especially for the worst-case performance. The predicate-based estimation approach managed to avoid the pitfalls of the formulaic approach through its overshooting cardinality estimates, which resulted in less processing needed to address various estimate shifts, and thus the CPU-seconds and GB-seconds were mostly identical to baseline, ignoring the experiment without added network latency, that failed to execute most of the queries and this exhibited abnormally low values for these metrics.

Applying client-side rate limiting reduced the system resource consumption by the query engine considerably, cutting CPU-seconds by 50...60% and GB-seconds by 60...75% across the board. This supports the observation of the engine dedicating its resources to link traversal and document parsing, even though this is clearly unnecessary and hinders performance. This suboptimal resource allocation also causes the engine to use more system resources than it needs, which directly translates to costs when running the query engine.

Additionally, through manual execution of several queries to better understand the engine behaviour, it was observed that the engine continues processing its internal link queue even after query execution is over. The link queue is the internal link buffer, that links to be followed are extracted into during the parsing of downloaded documents, and where further links to follow are pulled from. This link queue is not emptied upon query execution termination, and the engine continues pulling links from it until it is exhausted, independent of the query execution state, unless an engine error occurs. This behaviour causes the engine to perform unnecessary processing after finishing a query.

6.3. Network utilisation

The *network ingress* and *egress* capture the data download and upload on an experiment level, implemented in *jbr.js* similarly to the *GB-seconds* and *CPU-seconds* metrics, and thus also comparable across experiments, but impossible to associate with individual queries. These metrics allow for the analysis of data transfer, to help understand how much or little data is needed to actually answer a given query, and to identify configurations that use more network resources that they need to answer the query. These metrics also help understand the relative cost of query execution in environments where billing is based on network utilisation.

From the results in Table 3, one can notice how all the experiments turn in similar network utilisation numbers, downloading around 25...31 GB worth of data, while sending out requests worth 1...2 GB in total. Although this

may not be much for the entire benchmark execution with replications included, the rate-limited results demonstrate how most of this data downloaded is unnecessary, and is also a product of the query engine prioritising data access over query processing, and thus ending up doing more link traversal than is necessary to answer the queries. The problem is further excacerbated by the engine continuing to empty the link queue after query execution is over. The rate-limited experiments download data in the range of 5...10 GB, around 1/5...1/3 of the default setup, and sends out requests worth less than 0.5 GB, amounting to less than 1/2...1/4 of the default. This not only increases the cost of traversal for environments where network usage is factored into the service fees, but also unnecessarily elevates the costs for hosting data, when the query engine pulls more data from data providers than it actually needs, making the engine an inconsiderate client from the data servers' perspective.

7. Discussion

Although the average query execution times remained close to baseline, it was possible to achieve some considerable improvements through restart-based query planning. Unlike in our previous work, through a variety of optimisations both on engine-level and in our implementation, the formulae from Hagedorn et al. [28] did not perform the best, due to the necessary VoID description data becoming available slowly over the duration of the execution, and the formulae being so thorough in their estimations, that the final cardinalities changed several times, prompting the restart-based technique to spend its execution time improvements on the restart overhead. This would directly confirm Hypothesis 3, were it not for the predicate-based estimator, that manages to avoid the worst-case performance regressions of the formulaic approach, and improve performance relative to baseline. Thus, we are required to reject this hypothesis. We are also forced to reject Hypothesis 4, not because of the formulae themselves, but because of the nature of link traversal, where the information needed for the formulae becomes available in such small chunks that the formulae appear to produce unstable estimates over time.

With some interval-based evaluations also performing better than baseline, we are inclided to accept Hypothesis 1, as in our previous work, under the assumption that more robust cost-benefit estimation will help avoid further worst-case regressions and ultimately improve the average performance. We will also need to accept Hypothesis 2, due to uniform evaluation intervals generally performing inconsistently, underlining the importance of *reactive* query plan evaluation over a polling-style one.

The most interesting observation from this work has been the impact of rate limiting. Unlike in the work from [12], we chose to apply client-side rate limiting, to bring our experiments closer to real-world conditions, where servers refuse to handle hundreds of HTTP requests per second from an individual client over extensive periods of time. The initial assumption was that this would marginalise the impact of local processing, and align more closely with related work where this was shown to be the case [9], yet this did not happen. The rate-limited imposed on the engine forced it to dedicate more time to local query processing, at the expensive of data access, resulting in smoother result arrival rates over time, as well as better query performance with the predicate-based cardinality estimator. Thus, we reject Hypothesis 5.

Applying a uniform network latency also did not universally marginalise the impact of local query processing, with client-side optimisations still providing measurable improvements at 50-millisecond network latency. Thereby, we reject Hypothesis 6, due to being too universal. Applying a network latency of 100 milliseconds, however, did negate the impact of local optimisations, when the time spent on data access became too high relative to the overall query execution time, as the average total time spent on HTTP requests even exceeded the average query execution time. This serves to prove the point of [9] about data access over networks dwarfing the impact of local query processing, but also demonstrates how there is a window in the latency values where local processing can have a measurable impact. With 50-millisecond network latency fitting within this window, most applications operating within a single continent where latencies are expected to fall within this window should still be able to achieve measurable improvements, as outlined in the results.

Outside query performance, the use of system resources stood out in the experiment results. Due to the query engine being unable to manage the resource allocation between data acquisition and local query processing, it ends up dedicating unnecessary amounts of resources to data access, which not only reduces the relative impact of local query processing, but also reduces the overall user-perceived query performance. Furthermore, by consuming more

resources than it actually needs, the engine also makes itself more expensive to run, and prevents its own use in resource-constrained environments. Beyond the engine itself, this behaviour, together with the forced exhaustion of the link queue after execution, also resulted in more network traffic than necessary, notably 3...5× the amount of data download relative to what is actually needed, making the engine a poor client from the perspective of data providers upon whom this overhead is ultimately imposed in full. Future work on the engine itself should thereby prioritise addressing this issue with suboptimal resource allocation, to make the engine not only more performant with the same set of hardware, but also able to run with less resources, such as in edge computing scenarios.

8. Conclusions

The results from our experiments lead us to conclude that query performance *can* be improved through client-side adaptive techniques, over a heuristics-based approach. This is our answer to Question 1, and the primary conclusion of our work. Related work placed a theoretical oracle at roughly twice the performance of a heuristics-based approach in the best-case scenario [12], and thereby defined the upper bound of what should be possible through client-side optimisations. Our work here provides a lower bound to complement this, with an average of up to 15% improvements attainable through a restart-based approach. The current and future state-of-the-art techniques should thereby land in this 15...50% improvement window of client-side optimisations purely based on query planning.

The secondary goal of this work, specifically our extension to the original experiments, was to establish an understanding of the impact of network overhead. Related work demonstrated the marginal impact of local query processing [9], through a set of random query plans with identical performance, due to the data access costs dwarfing any impact of local query planning. Within this work, we applied request rate limiting to simulate real-world client-server interactions, and repeated the experiments with 50 and 100 milliseconds of additional latency, to simulate more realistic scenarios. Our discoveries indicate a window below 100 milliseconds, where network latency does not completely dwarf the query plan, but latencies close to and above 100 milliseconds appear to behave as described in related work [9]. This answers Question 2, in that rate limiting or network latency do not completely erase the impact of client-side optimisations, but with high enough latencies or aggressive enough rate limits, they definitely will, which also makes sense intuitively.

Overall, our experiments show potential in client-side optimisations, even in environments with realistic levels of network overhead. Through state-of-the-art adaptive optimisation techniques, alongside state-of-the-art cost-benefit analysis, tangible performance improvements can be attained, and future work in this direction should continue. Even in link traversal scenarios, purpose-built client-side optimisation techniques should provide significant improvements. To enable decentralisation initiatives such as Solid to succeed, performant query engine abstraction layers will be needed, to assist developers in producing services that take advantage of the benefits of decentralised data storage solutions.

Our results also highlight several shortcomings in the Comunica query engine itself, that when properly addressed, should enable query engine performance in the context of interactive applications, lower system resource consumption, and also better client behaviour towards data publishers. For example, decoupling link traversal and document parsing from the query processing should result in both performing better, provided there is no significant architecture-imposed overhead introduced that negates the performance benefits. While this is an engineering issue, it underlines the importance of the engineering aspect alongside the algorithm design, where implementation oversights can inadvertently introduce performance bottlenecks that hide of negate impact of various algorithms.

We believe the future of application development on top of decentralisation initiatives such as Solid depend on client-side query engine abstraction layers, and these abstraction layers need further research in client-side query optimisation to reach their peak performance, as well as implementations of these optimisations that do not suffer from engineering-introduced architecture overhead or implementation limitations, so that developers depending on them can deliver software and services that perform up to the expectations of their users.

Acknowledgements

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

References

[1] R. Verborgh, Re-decentralizing the Web, for good this time, in: *Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web*, 2023, pp. 215–230.
[2] R. Taelman, Towards Applications on the Decentralized Web using Hypermedia-driven Query Engines, *ACM SIGWEB Newsletter* (2024).
[3] J. Hanski, S. Van Braeckel, R. Taelman and R. Verborgh, Link Traversal over Decentralised Environments using Restart-Based Query Planning, in: *International Conference on Web Engineering*, 2025.
[4] O. Hartig, C. Bizer and J.-C. Freytag, Executing SPARQL queries over the web of linked data, in: *The Semantic Web-ISWC 2009: 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings 8*, 2009, pp. 293–309.
[5] T. Berners-Lee, Linked Data - Design Issues, 2006. http://www.w3.org/DesignIssues/LinkedData.html.
[6] O. Hartig and A. Langegger, A database perspective on consuming linked data on the web, *Datenbank-Spektrum* **10** (2010), 57–66.
[7] O. Hartig, Zero-knowledge query planning for an iterator implementation of link traversal based query execution, in: *Extended Semantic Web Conference*, 2011, pp. 154–169.
[8] G. Halasz Frank, Reflections on NoteCards: Seven issues for the next generation of hypermedia systems, *Communications of the ACM* **31**(7) (1988), 836–852.
[9] O. Hartig and M.T. Özsu, Walking without a map: optimizing response times of traversal-based linked data queries (extended version), *arXiv preprint arXiv:1607.01046* (2016).
[10] Y.E. Ioannidis and S. Christodoulakis, On the propagation of errors in the size of join results, in: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, 1991, pp. 268–277.
[11] L. Heling and M. Acosta, Robust query processing for linked data fragments, *Semantic Web* **13**(4) (2022), 623–657.
[12] R. Taelman and R. Verborgh, Link Traversal Query Processing Over Decentralized Environments with Structural Assumptions, in: *International Semantic Web Conference*, 2023, pp. 3–22.
[13] A. Deshpande, Z. Ives, V. Raman et al., Adaptive query processing, *Foundations and Trends in Databases* **1**(1) (2007), 1–140.
[14] R.L. Cole and G. Graefe, Optimization of dynamic query evaluation plans, in: *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 1994, pp. 150–160.
[15] S. Babu, P. Bizarro and D. DeWitt, Proactive re-optimization, in: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005, pp. 107–118.
[16] K. Eurviriyanukul, N.W. Paton, A.A. Fernandes and S.J. Lynden, Adaptive join processing in pipelined plans, in: *Proceedings of the 13th International Conference on Extending Database Technology*, 2010, pp. 183–194.
[17] R. Avnur and J.M. Hellerstein, Eddies: Continuously adaptive query processing, in: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 261–272.
[18] M. Acosta and M.-E. Vidal, Networks of Linked Data Eddies: An Adaptive Web Query Processing Engine for RDF Data, in: *The Semantic Web - ISWC 2015*, 2015, pp. 111–127. ISBN 978-3-319-25007-6.
[19] A. Deshpande and J.M. Hellerstein, Lifting the burden of history from adaptive query processing, in: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, 2004, pp. 948–959.
[20] C. Bizer and A. Schultz, Benchmarking the performance of storage systems that expose SPARQL endpoints, in: *Proc. 4 th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, Citeseer, 2008, p. 39.
[21] 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*, 2015, pp. 619–630.
[22] S. Capadisli, T. Berners-Lee, R. Verborgh and K. Kjernsmo, Solid Protocol 0.10.0, W3C Community Group Technical Report, W3C, 2022, https://solidproject.org/TR/2022/protocol-20221231.
[23] J. Arwe, A. Malhotra and S. Speicher, Linked Data Platform 1.0, W3C Recommendation, W3C, 2015, https://www.w3.org/TR/2015/REC-ldp-20150226/.
[24] A. Coburn, L. Debackere and E. Prud'hommeaux, Linked Web Storage Working Group Charter, W3C Working Group Charter, W3C, 2024, https://www.w3.org/2024/09/linked-web-storage-wg-charter.
[25] R. Eschauzier, R. Taelman and R. Verborgh, The R3 Metric: Measuring Performance of Link Prioritization during Traversal-based Query Processing, in: *Proceedings of the 16th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2024.
[26] 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*, IEEE, 2011, pp. 984–994.
[27] 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*, 2008, pp. 595–604.

[28] S. Hagedorn, K. Hose, K.-U. Sattler and J. Umbrich, Resource planning for SPARQL query execution on data sharing platforms, in: *Proceedings of the 5th International Conference on Consuming Linked Data-Volume 1264*, 2014, pp. 49–60.
[29] R. Cyganiak, K. Alexander, J. Zhao and M. Hausenblas, Describing Linked Datasets with the VoID Vocabulary, W3C Note, W3C, 2011, https://www.w3.org/TR/2011/NOTE-void-20110303/.
[30] S. Chaudhuri, R. Kaushik, A. Pol and R. Ramamurthy, Stop-and-restart style execution for long running decision support queries, in: *Proceedings of the 33rd international conference on Very large data bases*, 2007, pp. 735–745.
[31] P. Chalasani, S. Jha, O. Shehory and K. Sycara, Query restart strategies for web agents, in: *Proceedings of the second international conference on Autonomous agents*, 1998, pp. 124–131.
[32] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C Recommendation, W3C, 2013, https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.
[33] T. Urhan and M.J. Franklin, Xjoin: A reactively-scheduled pipelined join operator, *Bulletin of the Technical Committee on* (2000), 27.
[34] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo and E. Ruckhaus, ANAPSID: an adaptive query processing engine for SPARQL endpoints, in: *The Semantic Web–ISWC 2011: 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I 10*, 2011, pp. 18–34.
[35] M. Kitsuregawa, H. Tanaka and T. Moto-Oka, Application of hash to data base machine and its architecture, *New Generation Computing* **1** (1983), 63–74.
[36] J. Van Herwegen and R. Verborgh, The Community Solid Server: Supporting research & development in an evolving ecosystem, *Semantic Web* **15**(6) (2024), 2597–2611.
[37] R. Taelman, J. Van Herwegen, M. Vander Sande and R. Verborgh, Comunica: A Modular SPARQL Query Engine for the Web, in: *Proceedings of the 17th International Semantic Web Conference*, 2018, pp. 239–255.
[38] M. Acosta, M.-E. Vidal and Y. Sure-Vetter, Diefficiency metrics: measuring the continuous efficiency of query processing approaches, in: *The Semantic Web–ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part II 16*, 2017, pp. 3–19.

Experiment	Join order based on	Cardinality estimator	Plan evaluation	Restart limit	Network latency (ms)	Rate limited
baseline-0ms	heuristics without cardinality	-	-	-	-	no
	heuristics without cardinality	-	-	-	-	yes
baseline-50ms	heuristics without cardinality	-	-	-	50	no
	heuristics without cardinality	-	-	-	50	yes
baseline-100ms	heuristics without cardinality	-	-	-	100	no
	heuristics without cardinality	-	-	-	100	yes
overhead-hagedorn-0ms	cardinality estimate	formulae	-	-	-	no
	cardinality estimate	formulae	-	-	-	yes
overhead-hagedorn-50ms	cardinality estimate	formulae	-	-	50	no
	cardinality estimate	formulae	-	-	50	yes
overhead-hagedorn-100ms	cardinality estimate	formulae	-	-	100	no
	cardinality estimate	formulae	-	-	100	yes
overhead-predicate-0ms	cardinality estimate	predicate	-	-	-	no
	cardinality estimate	predicate	-	-	-	yes
overhead-predicate-50ms	cardinality estimate	predicate	-	-	50	no
	cardinality estimate	predicate	-	-	50	yes
overhead-predicate-100ms	cardinality estimate	predicate	-	-	100	no
	cardinality estimate	predicate	-	-	100	yes
restart-update-hagedorn-0ms	cardinality estimate	formulae	on estimate update	-	-	no
	cardinality estimate	formulae	on estimate update	-	-	yes
restart-update-hagedorn-50ms	cardinality estimate	formulae	on estimate update	-	50	no
	cardinality estimate	formulae	on estimate update	-	50	yes
restart-update-hagedorn-100ms	cardinality estimate	formulae	on estimate update	-	100	no
	cardinality estimate	formulae	on estimate update	-	100	yes
restart-update-predicate-0ms	cardinality estimate	predicate	on estimate update	-	-	no
	cardinality estimate	predicate	on estimate update	-	-	yes
restart-update-predicate-50ms	cardinality estimate	predicate	on estimate update	-	50	no
	cardinality estimate	predicate	on estimate update	-	50	yes
restart-update-predicate-100ms	cardinality estimate	predicate	on estimate update	-	100	no
	cardinality estimate	predicate	on estimate update	-	100	no
restart-100ms-hagedorn	cardinality estimate	formulae	100 ms intervals	-	-	no
	cardinality estimate	formulae	100 ms intervals	-	-	yes
restart-100ms-hagedorn-once	cardinality estimate	formulae	100 ms intervals	1	-	no
	cardinality estimate	formulae	100 ms intervals	1	-	yes
restart-100ms-predicate	cardinality estimate	predicate	100 ms intervals	-	-	no
	cardinality estimate	predicate	100 ms intervals	-	-	yes
restart-100ms-predicate-once	cardinality estimate	predicate	100 ms intervals	1	-	no
	cardinality estimate	predicate	100 ms intervals	1	-	yes
restart-1000ms-hagedorn	cardinality estimate	formulae	1,000 ms intervals	-	-	no
	cardinality estimate	formulae	1,000 ms intervals	-	-	yes
restart-1000ms-hagedorn-once	cardinality estimate	formulae	1,000 ms intervals	1	-	no
	cardinality estimate	formulae	1,000 ms intervals	1	-	yes
restart-1000ms-predicate	cardinality estimate	predicate	1,000 ms intervals	-	-	no
	cardinality estimate	predicate	1,000 ms intervals	-	-	yes
restart-1000ms-predicate-once	cardinality estimate	predicate	1,000 ms intervals	1	-	no
	cardinality estimate	predicate	1,000 ms intervals	1	-	yes
restart-10000ms-hagedorn	cardinality estimate	formulae	10,000 ms intervals	-	-	no
	cardinality estimate	formulae	10,000 ms intervals	-	-	yes
restart-10000ms-hagedorn-once	cardinality estimate	formulae	10,000 ms intervals	1	-	no
	cardinality estimate	formulae	10,000 ms intervals	1	-	yes
restart-10000ms-predicate	cardinality estimate	predicate	10,000 ms intervals	-	-	no
	cardinality estimate	predicate	10,000 ms intervals	-	-	yes
restart-10000ms-predicate-once	cardinality estimate	predicate	10,000 ms intervals	1	-	no
	cardinality estimate	predicate	10,000 ms intervals	1	-	yes

Table 2

The complete description of the test cases, to map the experiment names to their engine configurations. The experiments were repeated with client-side rate limiting and without it. The formulaic cardinlaity estimator uses the formulae from Hagedorn et al. [28], and the predicate-based estimator uses the predicate count from VoID descriptions. The restart limit restricts the number of query plan restarts.

Experiment	Duration (s) min	Duration (s) avg	Duration (s) max	First result (s) min	First result (s) avg	First result (s) max	Last result (s) min	Last result (s) avg	Last result (s) max	*dieff@full* min	*dieff@full* avg	*dieff@full* max	HTTP requests	CPU-sec (%)	GB-sec	Network (GB) ingr.	Network (GB) egr.	Queries finished
baseline-0ms	0.23	0.49	1.39	0.06	0.32	1.52	0.06	0.38	1.52	0.03	1.57	32.00	62	213,936	53,865	31	2	55 / 75
	0.55	1.09	3.40	0.07	0.63	5.44	0.07	0.78	5.44	0.04	3.28	24.70	62	84,232	14,821	7	0	60 / 75
baseline-50ms	0.50	0.82	1.73	0.28	0.63	1.86	0.28	0.70	1.86	0.14	2.05	23.54	62	203,679	41,663	29	2	54 / 75
	0.74	1.64	5.66	0.27	1.13	6.43	0.27	1.26	6.43	0.14	3.14	23.59	62	86,675	14,518	6	0	59 / 75
baseline-100ms	0.76	1.09	2.16	0.47	0.87	2.44	0.47	0.93	2.45	0.24	1.63	10.70	62	217,614	44,029	29	1	57 / 75
	0.96	1.75	4.87	0.46	1.25	5.53	0.46	1.38	5.53	0.23	2.93	19.48	62	86,566	14,537	6	0	61 / 75
overhead-hagedorn-0ms	0.24	0.55	1.79	0.06	0.36	2.28	0.06	0.44	2.28	0.03	1.94	**20.81**	62	**193,236**	**38,892**	**27**	**1**	**57 / 75**
	0.56	1.13	3.64	0.07	0.67	**4.83**	0.08	0.80	**4.83**	0.04	3.32	**19.19**	62	**83,170**	**13,921**	**6**	0	**66 / 75**
overhead-hagedorn-50ms	0.50	0.86	2.22	0.26	0.67	2.39	**0.26**	0.74	2.39	**0.13**	2.31	24.69	62	**198,438**	**40,711**	**27**	1	**61 / 75**
	0.72	1.67	6.02	0.27	1.14	7.02	0.27	1.28	7.02	**0.13**	3.46	**22.65**	62	**84,783**	15,009	**5**	0	**64 / 75**
overhead-hagedorn-100ms	0.77	1.10	2.31	0.47	0.89	**2.43**	0.47	0.96	**2.43**	**0.23**	2.69	32.33	62	**193,274**	**36,108**	**25**	1	**62 / 75**
	0.99	1.90	6.27	0.47	1.37	7.35	0.47	1.52	7.35	0.24	3.53	28.78	62	**81,989**	**14,013**	**5**	0	**64 / 75**
overhead-predicate-0ms	**0.22**	0.54	1.89	**0.05**	0.37	2.19	**0.05**	0.44	2.19	**0.02**	2.09	**20.97**	62	**194,024**	**37,624**	**27**	**1**	**57 / 75**
	0.56	1.17	4.11	**0.06**	0.69	5.97	0.08	0.83	5.97	0.04	3.48	**24.25**	62	84,638	**14,701**	**6**	0	**65 / 75**
overhead-predicate-50ms	**0.50**	0.85	2.15	0.27	0.66	2.33	**0.27**	0.74	2.33	**0.13**	2.39	**22.89**	62	**190,224**	**36,636**	**26**	1	**61 / 75**
	0.72	**1.61**	5.78	0.26	1.13	7.09	**0.26**	1.25	7.09	**0.13**	3.21	**20.81**	62	**82,441**	15,614	**5**	0	**65 / 75**
overhead-predicate-100ms	0.77	1.12	2.53	0.46	0.91	2.71	**0.46**	0.98	2.71	**0.23**	2.38	33.04	62	**200,461**	**42,964**	**25**	1	**60 / 75**
	0.98	1.92	6.61	0.46	1.40	7.65	0.46	1.54	7.65	0.23	3.53	23.85	62	**81,140**	14,568	**5**	0	**65 / 75**
restart-update-hagedorn-0ms	0.23	0.84	6.32	**0.05**	0.68	6.37	0.07	0.75	6.37	0.03	1.87	**16.92**	62	290,847	72,603	**27**	2	**57 / 75**
	0.57	1.66	8.78	0.07	1.19	12.60	0.08	1.34	12.60	0.04	3.74	**20.63**	62	103,610	17,918	8	0	**62 / 75**
restart-update-hagedorn-50ms	**0.47**	1.08	4.11	0.26	0.91	6.68	**0.26**	0.98	6.68	**0.13**	3.33	28.85	62	229,516	49,187	**27**	2	**62 / 75**
	0.72	2.75	16.42	0.27	2.27	18.14	0.27	2.41	18.14	0.14	3.96	**18.01**	62	103,370	18,571	7	0	**63 / 75**
restart-update-hagedorn-100ms	0.77	1.60	6.13	0.46	1.38	8.52	0.47	1.45	8.52	**0.23**	2.79	28.81	62	246,479	53,031	**26**	2	**59 / 75**
	0.96	3.45	21.26	0.46	2.94	37.94	0.46	3.08	37.94	0.23	4.02	**18.97**	62	102,832	17,969	7	0	**64 / 75**
restart-update-predicate-0ms	0.24	**0.44**	**0.90**	**0.05**	**0.27**	**1.18**	0.08	**0.35**	**1.18**	0.04	2.27	**25.60**	62	**54,515**	**14,316**	**4**	**0**	25 / 75
	0.52	**0.99**	**2.59**	0.07	**0.53**	**3.19**	0.08	**0.68**	**3.19**	0.04	**3.20**	**19.95**	62	144,616	39,377	9	0	59 / 75
restart-update-predicate-50ms	**0.50**	**0.70**	1.15	0.26	**0.53**	**1.17**	**0.26**	**0.60**	**1.17**	**0.13**	2.38	24.54	62	205,765	45,913	29	2	**57 / 75**
	0.76	**1.51**	4.45	0.26	**1.01**	**6.21**	0.27	1.15	6.22	0.14	3.37	**22.85**	62	93,123	16,325	8	0	59 / 75
restart-update-predicate-100ms	0.77	**0.97**	1.48	0.46	**0.77**	**1.79**	0.47	0.84	**1.79**	0.24	2.39	34.08	62	**189,849**	**41,207**	**27**	2	57 / 75
	1.00	1.78	5.38	0.47	1.27	6.22	0.48	1.41	6.22	0.24	3.29	**16.36**	62	93,941	16,026	8	0	59 / 75
restart-100ms-hagedorn	0.24	0.88	6.37	0.06	0.71	6.55	0.07	0.79	6.55	0.04	2.10	**24.94**	62	312,396	79,242	**27**	2	**57 / 75**
	0.54	1.68	8.20	0.07	1.24	11.40	0.08	1.37	11.40	0.04	3.62	**23.50**	62	112,351	20,056	8	0	**61 / 75**
restart-100ms-hagedorn-once	**0.21**	0.83	6.23	0.06	0.67	6.46	0.06	0.74	6.46	0.03	2.25	**23.75**	62	243,300	**52,867**	**26**	2	**58 / 75**
	0.59	1.60	7.40	**0.06**	1.13	13.61	0.08	1.26	13.62	0.04	3.41	**18.00**	62	106,678	17,897	8	0	**64 / 75**
restart-100ms-predicate	**0.21**	0.56	2.35	**0.05**	0.38	3.91	**0.05**	0.45	3.91	0.03	1.90	**18.25**	88	229,757	**51,323**	**27**	2	52 / 75
	0.52	1.11	3.80	0.07	0.64	**5.13**	0.08	**0.77**	**5.13**	0.04	**3.09**	**14.88**	62	101,290	18,215	8	0	54 / 75
restart-100ms-predicate-once	**0.21**	**0.43**	**0.90**	0.06	**0.26**	**1.22**	0.06	**0.33**	**1.22**	0.03	1.77	32.60	62	240,729	57,610	**27**	2	54 / 75
	0.53	1.10	3.56	0.07	0.64	6.44	0.08	**0.77**	6.44	0.04	3.46	**17.66**	62	109,635	17,686	8	0	**64 / 75**
restart-1000ms-hagedorn	**0.22**	0.53	1.62	0.06	0.36	1.87	0.07	0.44	1.87	0.03	2.19	**23.84**	62	224,702	**46,207**	**26**	2	55 / 75
	0.52	**1.07**	3.25	**0.05**	**0.62**	**4.96**	**0.05**	**0.76**	**4.96**	**0.02**	3.32	**24.03**	62	110,697	19,560	8	0	60 / 75
restart-1000ms-hagedorn-once	**0.21**	0.53	1.63	**0.04**	0.36	1.79	**0.05**	0.43	1.79	0.03	2.05	**24.57**	62	242,955	**51,566**	**25**	2	**56 / 75**
	0.54	1.16	4.13	**0.06**	0.70	**4.93**	**0.06**	0.84	**4.93**	**0.03**	3.35	**18.45**	62	108,675	19,990	8	0	**61 / 75**
restart-1000ms-predicate	**0.20**	0.52	1.54	**0.05**	0.36	1.60	**0.05**	0.43	1.60	0.03	2.10	**26.33**	62	265,700	65,916	**27**	2	**59 / 75**
	0.58	1.32	3.52	0.07	0.83	14.80	0.07	0.98	14.80	0.04	3.79	**23.43**	66	98,772	18,973	8	0	57 / 75
restart-1000ms-predicate-once	**0.21**	0.53	1.62	**0.04**	0.35	1.95	0.07	0.43	1.95	0.03	1.88	**22.63**	62	236,727	54,767	**28**	2	**58 / 75**
	0.58	1.12	3.42	**0.06**	0.65	**4.79**	**0.06**	0.78	**4.79**	**0.03**	3.49	29.34	62	101,542	17,009	8	0	**62 / 75**
restart-10000ms-hagedorn	**0.21**	0.57	1.71	**0.04**	0.40	3.65	**0.04**	0.47	3.65	**0.02**	1.68	**21.83**	62	220,080	**48,758**	**30**	2	**58 / 75**
	0.57	1.18	3.85	0.07	0.67	**4.84**	0.08	0.81	**4.84**	0.04	3.76	26.98	62	100,986	17,415	9	0	**63 / 75**
restart-10000ms-hagedorn-once	**0.22**	0.55	1.89	**0.05**	0.37	1.97	0.07	0.45	1.97	0.03	2.30	**28.87**	62	248,495	59,338	31	2	**57 / 75**
	0.54	1.11	3.71	0.07	0.66	**4.34**	0.07	0.79	**4.34**	**0.03**	3.49	**19.75**	62	111,166	19,898	9	0	**62 / 75**
restart-10000ms-predicate	**0.21**	0.52	1.79	**0.05**	0.36	1.83	0.06	0.43	1.83	0.03	2.08	**24.25**	62	237,755	55,339	**30**	2	**59 / 75**
	0.57	1.13	3.61	0.07	0.65	5.70	0.07	0.79	5.70	0.04	3.64	**23.00**	62	101,088	16,615	9	0	**64 / 75**
restart-10000ms-predicate-once	**0.21**	0.53	1.81	**0.05**	0.37	1.87	**0.05**	0.45	1.87	0.03	2.32	**31.10**	62	284,735	71,381	31	2	**58 / 75**
	0.59	1.19	3.63	0.08	0.68	**5.25**	0.08	0.83	**5.25**	0.04	3.50	**17.77**	62	102,244	17,397	9	0	**63 / 75**

Table 3

Overview of the benchmark results, for both the default HTTP request behaviour, and with client-side rate limiting applied. Measurements better than their respective baselines are highlighted in **bold**. The use of client-side rate limiting reduces absolute query performance, and increases *dieff@full*, but allows more queries to succeed, while also considerably reducing data transfer over network, as well as the relative CPU-seconds used by the query engine, as well as the GB-seconds, measured as an integral of memory consumption over execution time. The HTTP request count reported is the number of requests done by the time the query finished.