Semantic Web 1 (0) 1–4
IOS Press

Measuring the relative impact of network overhead and client-side query optimisation in link traversal over Linked Data documents

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

Abstract. Data decentralisation initiatives, while addressing issues around regulatory compliance and barriers to entry into data-driven markets, also introduce new technical challenges with regard to data access. These challenges are imposed upon the data access abstraction layer, such as client-side query engines, that become responsible for performant data access. Especially in the context of interactive applications, user-perceived sluggishness can inhibit the adoption of applications built on decentralised data, and, by extension, the decentralisation initiatives themselves, should they appear infeasible for such use cases. The performance of the data access layer is the combination of data retrieval over the network and the processing of this data locally. 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 measure the impact of network overhead on query execution over decentralised Linked Data documents, when combined with an example restart-based adaptive query planning approach, using the Solid ecosystem as an example environment. Through the addition of rate-limiting and network latency simulation, we are able to demonstrate how client-side query optimisations that provided considerable improvements under optimal network conditions in prior work are unable to produce measurable improvements under more realistic conditions. Our work illustrates the importance of using realistic benchmarking environments, and highlights the importance of future work on reducing or circumventing the network overhead as a whole in link traversal.

Keywords: Solid, Linked Data, Link Traversal

1. Introduction

With ever-increasing numbers of users, and growing amounts of data alongside increasing legislative requirements, centralised approaches to data management are facing challenges [1]. Beyond compliance with privacy-related legislation, practical issues regarding service interoperability and barriers to entry into data-driven markets arise, where users of different existing services are unable to interact with each other, such as between social media platforms, or where new service providers need to collect and manage considerable amounts of data to offer competitive services, such as advertising or personalisation services [2]. These challenges are addressed by alternative, decentralised and distributed approaches to data management. In the social networking space, for example, the ActivityPub [3] specification, based on Linked Data [4], has been used within the Mastodon [5] initiative to provide an alternative to microblogging services. Within the government sector, the Solid [1] initiative has been explored as a means to reduce data duplication, accelerate data sourcing for service delivery, and increase the transparency of

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

data management for the government and its citizens [6]. The Solid initiative itself builds on the W3C Linked Data Platform [7] specification, with further development taking place under the Linked Web Storage [8] specifications as the next iteration of read-write Linked Data storage on the Web [2].

In this work, we focus on the Solid initiative as a distributed Linked Data layer, upon which various applications could be built. The Solid [1] initiative is based on the concept of storing data in permissioned online data stores, with metadata and permissions declared as Linked Data, and with the data owner retaining control over these permissions. Through standardised access mechanisms, this single set of data could then be shared across services, and to update a data item across all those services, the user would only need to modify the entry within their pod. Service providers could also directly access service-relevant data from the user's personal data store, without the need to collect or manage such data themselves. 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.

Decentralisation, however, comes with its own set of challenges. Moving from a uniform, centralised data storage and access system to a decentralised, potentially non-uniform one may introduce a layer of complexity previously absent from the development of services, potentially leading to additional barriers to entry within the decentralised context. For example, to develop a service that uses 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 [9, 10]. As 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 only a minimal amount thereof. 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 these operations with sufficient user-perceived performance. The ideal, zero-knowledge approach to data discovery does, however, limit the scope for optimisation.

Within this work, we investigate the impact of rate limiting and network latency on client-side query optimisations for link traversal query execution within the Solid ecosystem [1]. This work is an extension of our earlier investigation [11], and the example restart-based adaptive query planning technique remains the same. 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 focus on the impact of network overhead previously absent from our experiments. Our earlier experiments used an experimental setup in which the engine sent HTTP requests to the server at an unrealistic rate, to the extent that the client would have been blocked by real remote servers due to excessive request rates. The earlier experiments also did not take into account proper network latencies, running locally with effectively zero latency. This does not reflect real-world conditions, and through this extension, we bring our results closer to practice, by taking into account realistic levels of network latency, as well as the impact of request rate limiting.

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) [12], which builds on the *Linked Data principles* [13]. Through widespread adoption of these principles, the World Wide Web enables a globally distributed dataspace in the form of *the Web of Linked Data* [14]. 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 [15].

The query solution mappings for traversal-based execution are considered complete under the specific *reachability semantics* bounded by these criteria, and the solution mappings between executions can only be compared under the same semantics [15]. This is due to traversal under *cNone* and *cAll* being able to follow different sets of links, and thus find different sets of data to execute the query over.

This is an automated, query-driven approach to browsing networks of interlinked data [16], 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 [17]. Using pipelined query execution, solution mappings can be produced as soon as possible, provided the query contains no blocking non-monotonic operations, such as `COUNT`, `GROUP BY`, `ORDER BY`, and others, that require all intermediate solution mappings before producing additional ones.

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 [18], also applies to decentralised scenarios where the data still has to be processed locally, and excessive numbers of intermediate solution mappings 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* [19], 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* [20] may be employed. However, the query plans produced by such heuristics may still differ greatly from the optimum [21].

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 [22], 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 the theoretical oracle in prior work [21], 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 [23, 24]. Measurable improvements have likewise been demonstrated through join operation reordering and algorithm replacement [25]. Other approaches include *data partitioning* methods, such as Eddies [26], 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) [27], 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 solution mappings, such as with the polymorphic bind and hash join operators [19], that swap between the two strategies at runtime. The Eddies have also been extended with essentially operator-internal techniques [28], 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 renders the local processing cost negligible [17]. 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 [29], adapted for several test networks with varying link structures, simulating an online e-commerce environment.

On the other hand, considerable performance improvements have been demonstrated [21] over a social forum dataset, adapted for a distributed link traversal scenario from the LDBC social network benchmark dataset [30], 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 query engine and Solid 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, where the communication between the query engine and the Solid server exhibits realistic levels of latency and rate-limiting, with the expectation that the overall performance impact will range from negligible to 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 [31]. The contents of pods are exposed as a tree-like linking structure consisting of containers and resources, using the Linked Data Protocol (LDP) [7]. 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 [32], which may eventually result in data sources outside the Solid ecosystem sharing some aspects of its convenient means of exposing data for machine processing.

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

This constrained and well-defined means of publishing data in Solid pods has been shown to increase the relative impact of query planning [21], when sufficient structure exists to allow efficient location of query-relevant data, shifting the bottleneck from data access to the local processing needed to produce the solution mappings for the query. Further reductions in network overhead have been demonstrated through the use of link prioritisation during traversal [33], as well as through pruning of links to query-irrelevant documents by [34]. 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 and the specific join algorithms – between them, in an effort to minimise the number of intermediate solution mappings, thereby also minimising the amount of work needed to process the data. Prior work on the impact of cardinality estimation [18] has shown that, in query plans with multiple joins, errors in this estimation can cause exponential increases in the number of intermediate solution mappings. Although centralised storage solutions are often capable of pre-computing such information or providing estimates efficiently, such as through the use of *characteristics sets* [35], within decentralised scenarios this may not always be possible, yet the importance of accurate estimates remains high [21].

Thus, various purpose-built estimation techniques have to be applied in this context, such as *variable counting* [36], 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. [37], copied in Table 1, make use of the statistics offered in dataset descriptions published using the Vocabulary of Interlinked Datasets (VoID) [38], 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.

Table 1

Triple pattern cardinality estimation formulae from Hagedorn et al. [37], 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.

Triple pattern	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* exists	

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. [37] 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 similar factors, 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 [24].

3. Research questions

We seek to understand how realistic levels of network overhead affect traversal-based query execution over a decentralised environment, and how the relative impact of local query optimisations changes under such conditions, thereby allowing us to validate and challenge the results of related work where possible. We employ a selection of applicable query planning techniques from related work, and combine them with a restart-based mechanism for re-evaluating and restarting query plans to simulate adaptive approaches. We also introduce simulated network latency and rate limiting. The following research question forms the basis of our work:

Question 1. *Can overall query performance be improved through the use of client-side adaptive techniques, compared to heuristics-based zero-knowledge query planning, even in settings with rate-limiting and network latency?*

We derived the following hypotheses, inspired by prior work [11, 21], 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 solution mapping earlier, even in environments with rate-limiting and network latency.*

Hypothesis 2. *Using a static interval for plan evaluation and restart performs worse than evaluation and restart upon the discovery of new information about dataset characteristics. This is because blindly evaluating the join plan incurs unnecessary overhead.*

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. *The cardinality estimation formulae from [37] will outperform a simple predicate count-based estimation technique when VoID description metadata is used, highlighting the importance of cardinality estimation in achieving performance improvements.*

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

4. Client-side optimisation and simulation

This section outlines our approach to client-side query execution, partial query plan re-optimisation and re-execution, as well as network latency and rate-limiting simulations.

4.1. Client-side query execution

Within this work, we use Comunica [39], a modular client-side SPARQL query engine framework that provides a baseline link traversal implementation. This query engine has also been used to benchmark the relative impact of query plans compared to network overhead in related work [21], and is thus a good fit for our use case. The Comunica framework allows us to implement only the components needed to evaluate our specific approaches, ensuring a fair comparison between the different scenarios from a technical standpoint. The Comunica framework is available as open-source software[2], and supports SPARQL 1.1 and 1.2; however, within the context of this work, the benchmark queries only make use of a small subset of SPARQL 1.1, namely triple patterns, `OPTIONAL` clauses,

[2]https://github.com/comunica/comunica/tree/d3f4297

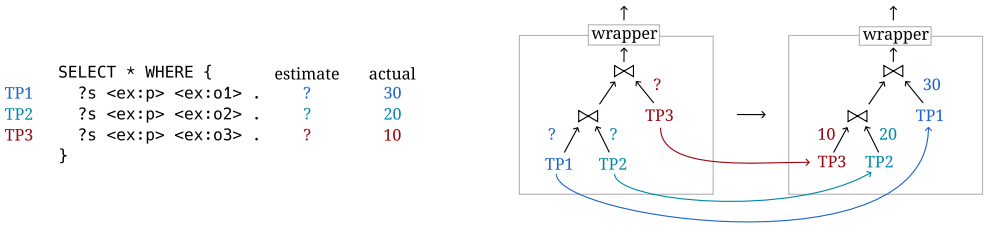

Fig. 1. Once the cardinalities are discovered, and the current plan is suboptimal due to TP1 producing the most solution mappings, the wrapper transparently restarts the query plan to place it higher up in the join tree, thus avoiding excessive intermediate solution mappings.

BIND, UNION, and aggregates. Thus, the focus of our work is limited to the subset of the language covered by the benchmark queries.

Within the Comunica engine, the link traversal and query execution aspects are decoupled: the traversal is based on an internal queue, and this queue is filled by extracting links directly from traversed documents, as well as the traversal seed URI, whereas the query execution takes place over an internal aggregated triple store for performance reasons. That is, when a triple is acquired through URI lookups and document parsing, it is sent to both the link extractor and the internal aggregate store. The engine uses a pull-based iterator approach [17] to query execution.

4.2. Partial query plan re-optimisation and re-execution

Within this work, to simulate adaptive query planning approaches, we employ a simple re-optimise and re-execute approach, where the engine re-optimises parts of the query plan and re-executes them from the beginning, should the current plan appear suboptimal. Our specific approach is to use a dummy inner join operator that performs this task, by acting as a transparent wrapper over the first inner join within the query plan. Within the context of our work, due to the benchmark queries detailed later, such a wrapper encapsulates everything under the projection, as the queries consist almost entirely of joins of triple patterns and unions.

The dummy operator receives the join entries as its input, it determines the current optimal join order using the active join ordering approach, forwards the entries to the actual join implementations of the engine, and proxies the output back one solution mapping at a time, as part of pipelined query execution. The dummy operator supports all types of joins that the base Comunica engine supports, including bind join (inner and left join), nested loop join (inner and left join), and symmetric hash join (inner and left join, as well as minus), to ensure non-blocking joins, while also supporting both bushy and left-deep plans. The dummy operator is therefore able to monitor the join order, compare it against an optimal one based on the cardinality available on the join entries, and re-execute the entire join operation when needed. This idea is illustrated in Fig. 1.

The dummy operator is able to transparently re-execute the join it was assigned, terminate the old execution, and continue proxying new solution mappings from the output of the new execution. The operator assumes that a join operation, when restarted, produces its full output again from the beginning, which is true in the Comunica engine implementation. Therefore, it internally keeps track of the count of solution mappings produced, and skips duplicates after re-execution based on this count tracking, allowing it to operate under bag semantics as required by the SPARQL specification [40]. While this record is maintained fully in memory within our implementation, and thus is unsuitable for use with queries producing more solution mappings than can be stored in system memory, practical solutions could look towards flushing it to disk as with XJoin [41] or agjoin [42], or splitting the record across several memory pools based on hashes such as with GRACE [43].

The dummy operator can be configured to compare the currently executing join order against the optimal one at fixed intervals or reactively, reflecting the hypotheses:

1. With *fixed interval* evaluations, the operator checks for changes in input entry cardinality estimates at the specified interval. 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. With *reactive* evaluations, the operator listens for input entry metadata updates, and upon updates checks for changes in cardinality estimates of the input entries. This approach allows for the evaluation of the query plan only when the information affecting query planning is updated, such as when a VoID dataset description is discovered.

Upon detecting changes in input entry cardinalities, the operator uses the chosen join ordering approach to determine the current optimal join order. Should this order be different from the currently executing one, the operator re-executes the join, and replaces the output stream with the new join output.

Our restart-based approach differs from stop-and-restart approaches [44], in that the join operation, forming a part of the total query plan, is terminated entirely under the dummy operator, and restarted from the beginning, with the sole goal of producing a better join order after the restart, with fewer intermediate solution mappings, and thus better performance. 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 [45], where the agent attempts to achieve higher performance by re-sending a query, in our case by re-running the join with a different ordering.

4.3. Join entry ordering

Within this work, we employ a set of join entry ordering techniques already present within the Comunica engine. These techniques are the following:

1. Zero-knowledge *heuristics* approach from Hartig [20], that does not rely on cardinality estimates, and instead focuses on the query itself, such as the dependencies between triple patterns with regard to their variables, and the prioritisation of patterns that should filter intermediate solution mappings.
2. Variable counting-based *selectivity* approach from Stocker et al. [36], that also does not rely on cardinality estimates, but performs fewer heuristics, with a focus on the relative selectivity of different triple patterns. Within the Comunica engine, this approach has been extended to support non-triple-pattern operations.
3. *Cardinality*-based greedy approach, where only the cardinality information is used, and the entries with the lowest cardinality are prioritised. With missing cardinality information, the original order is preserved.

These techniques are applied one by one, without any fallbacks to an alternative estimation technique. This will allow us to see how the join ordering approach itself affects local execution, and to ensure the cardinality-based test cases do not accidentally use a non-cardinality heuristic approach that would skew the results. The engine itself employs a greedy approach to join entry ordering, but also avoids cross products wherever possible, such as with more than two entries to a join, independent of cardinality estimates, to avoid worst-case scenarios where possible.

4.4. Triple pattern cardinality estimation

The cardinality-based join entry ordering approach 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 order every time. Ideally, any new cardinality estimate would be closer to the true cardinality value known only at the end of the processing. The cardinality estimates do not need to be precise, however, as they are used for prioritisation of operations relative to each other, and it is therefore enough for their relative magnitudes to reflect the relative magnitudes of the true cardinalities for the estimates to function as intended.

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 formulae from Hagedorn et al. [37], included in Table 1. For 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 a realistic and 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 see the impact of triple pattern cardinality estimation techniques, and to help avoid benchmarking triple pattern cardinality estimates by accident. The initial query plan with both cardinality estimation approaches, prior to execution, is produced without cardinalities for all triple patterns and query operations. Whenever new information becomes available, such as when a VoID dataset description is discovered by the engine during traversal, the cardinality estimates are updated immediately.

With nested operations, the cardinality changes are aggregated and propagated upwards within the metadata, so that estimates are available at higher levels of the query plan, as well, without any component needing a complete view of the entire query plan. The cardinality estimator uses recursion to find the lowest level operations, performs the cardinality estimation for these, and then constructs higher-level estimates while returning from its recursion. For example, for a `UNION` of two triple patterns, the estimator performs its estimates on the triple patterns, and then sets the `UNION` estimate to the sum of the pattern cardinalities. This approach is used for all other operations, as well. Notably, inner joins use a combination of input cardinality sums and products: the cardinalities of patterns with shared variables are added up, and then the summed values of detached pattern groups without variable overlap are multiplied by each other, to achieve a balance between worst-case cross product estimates and actual cardinalities.

4.5. Network latency and rate-limit simulation

With link traversal taking place over the HTTP protocol on the Web, we have decided to include two types of network overhead simulation, to account for the impact of more realistic network conditions that could be encountered in practice:

1. Request *rate limiting simulation*, to simulate scenarios where sending bursts of hundreds of HTTP requests concurrently is not possible due to server-side configuration.
2. Network *latency simulation*, specifically the round-trip time between the client and the server, to simulate scenarios where packet traversal across the Web takes longer than a sub-millisecond duration.

These will allow us to take into account more realistic network conditions and server-client interaction, and to better evaluate the scaling of client-side optimisations under more realistic scenarios. Both of these simulations were implemented engine-side within Comunica, and they are applied at the HTTP request level in the engine, just before sending actual network requests.

The rate limit simulation operates by matching client request rate to server response rate, on a server-by-server basis: 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 request interval i_n applied between request $n-1$ and n, where S is the constant smoothing multiplier and d_n is the delay between sending request n and receiving the response from the server to it.

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

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. Within our work here, the server is the Solid server serving Linked Data documents.

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. Within this work, we simulate added latencies of 0 and 100 milliseconds, with the 100 milliseconds value chosen based on the discoveries in [46], where an example round trip time between locations in Europe and North America was found to be in the order of 100 milliseconds.

5. Experimental setup

Within this section, we describe the benchmark used, explain the generation of VoID dataset descriptions, and outline the query engine configurations employed.

5.1. The SolidBench benchmark

The dataset and queries used for our experiments were generated using SolidBench[3], a benchmark to simulate a distributed social network use case across Solid pods using the LDBC SNB social network dataset from [30]. This benchmark has also been used in related work [21], 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. The dataset was served by the Community Solid Server [47], with access control disabled, turning the server into a Linked Data document server. The server exposes the contents of the pods as Linked Data Platform containers [7], by generating the tree-like container and document structure on the fly based on the underlying filesystem layout.

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, all based on the generated data with the intention of simulating real-world queries over the social network data. The queries instantiated using the complex template all failed, however, even with the baseline engine configuration, and have thus been excluded from this work. The remaining queries primarily consist of basic graph patterns, with `UNION`s, `OPTIONAL`s, and some `BIND` clauses. Thus, the queries primarily test joins, as well as the underlying data access mechanism by the engine.

5.2. Generation of VoID dataset descriptions

Within this work, we extended the RDF dataset fragmenter library used by the SolidBench dataset[4] to also produce VoID dataset descriptions [38] during benchmark data generation. The SolidBench data contains blank nodes by default, and we have also extended the generator with logic to convert them into full URIs under their respective documents, allowing all data to be associated with a pod based on its URI within the context of the benchmark. Thus, the benchmark produces both the data and the VoID descriptions for this data during the generation phase. The VoID descriptions were generated for each Solid pod, and summarise all data within a given pod. The data contained within a given Solid pod is therefore the dataset to which a VoID description applies. These descriptions were placed at the pod root as additional metadata, and the Community Solid Server served this metadata to a client that requested the pod root URI. Thus, the engine was able to discover these descriptions.

The social network data within different pods is divided into documents using three different approaches: by creation date, by creation location, or all in one document, but the data itself is still very similar across different pods, using the same RDF [4] vocabularies and having the same structure. Therefore, the overall VoID metadata between different pods is also very similar, and placing them at the pod root makes sense from this perspective. The cardinality estimation formulae will also produce similar values between different pods for this reason. While this is a limitation of the benchmark itself, for the purposes of our work this is acceptable, as the primary goal is to have some statistics available to support cardinality estimates, rather than to measure the handling of cardinality estimates over heterogeneous data.

5.3. The Solid reachability criteria and VoID description discovery

Within this work, the Comunica query engine was configured to use a hybrid reachability criterion, that is based upon the *cMatch* semantics, but extended to include Solid-specific predicates that are always followed, to ensure

[3] https://github.com/SolidBench/SolidBench.js/tree/fce8ff2
[4] https://github.com/SolidBench/rdf-dataset-fragmenter.js/tree/7d6c4ca

the engine traversed Solid pod LDP structures even when not explicitly included in the query. This is the same configuration as used in related work [21].

The benchmark queries contain the seed URIs for query execution, that is, the first URI to be looked up by the engine is found within the query it executes, and from this URI it then proceeds to traverse using the hybrid reachability criteria. The seed URIs for all but four query templates (short-4 to short-7) are WebID [48] links, and following the specifications, these contain `pim:storage` links to the pod roots for those WebIDs. When the engine follows this link, it automatically gets served the VoID description, and is able to parse it and update the cardinality estimates. Even for queries without WebID as the seed URI, the engine is able to find the pod root within a few hops, as all data eventually links to a WebID.

5.4. Query engine configurations

The experiments were executed using the following engine configurations:

1. Baseline test cases without restart-based mechanisms, repeated using different join ordering techniques, as well as without rate-limiting or latency simulation (no suffix), and with rate-limiting and added latency (*0ms* and *100ms*):
   i *baseline-traversal-heuristics*: Heuristics-based ordering.
   ii *baseline-variable-counting*: Variable counting-based ordering.
   iii *baseline-void-cardinality-hagedorn*: Cardinality-based ordering using formulae from Hagedorn et al.
   iv *baseline-void-cardinality-predicate*: Cardinality-based ordering using predicate-based estimates.
2. Interval-based join evaluation, performed every *n* milliseconds, with cardinality-based join ordering using different approaches:
   i *restart-polling-*n*ms-hagedorn-once*: Formulae from Hagedorn et al. with single restart limit.
   ii *restart-polling-*n*ms-hagedorn-unlimited*: Formulae from Hagedorn et al. without restart limit.
   iii *restart-polling-*n*ms-predicate-once*: Predicate-based estimates with single restart limit.
   iv *restart-polling-*n*ms-predicate-unlimited*: Predicate-based estimates without restart limit.
3. Reactive join evaluation, performed upon metadata updates, with cardinality-based join ordering using different approaches:
   i *restart-reactive-hagedorn-once*: Formulae from Hagedorn et al. with single restart limit.
   ii *restart-reactive-hagedorn-unlimited*: Formulae from Hagedorn et al. without restart limit.
   iii *restart-reactive-predicate-once*: Predicate-based estimates with single restart limit.
   iv *restart-reactive-predicate-unlimited*: Predicate-based estimates without restart limit.

The baseline configurations allow us to establish the relative impact of rate-limiting, network latency, and cardinality estimation approaches. The interval-based and reactive approaches, on their part, will provide measurements on the impact of join restarts. Together, these two will allow us to place the impact of restart-based techniques in perspective, and thus provide the floor for improvements attainable through adaptive techniques under different network conditions. Our modifications to the query engine are available as open source software[5].

5.5. Experiment execution

The experiments were all executed sequentially on the same machine[6], using the same jbr.js[7] benchmark orchestration tool as in related work[21]. The query engine and the Solid server were running in isolated Docker containers,

[5]https://github.com/surilindur/comunica-components/tree/a3027c6

[6]Hardware consisting of an AMD EPYC Milan 7003 series processor, 3200MHz DDR4 memory, and NVMe SSD for storage, running Fedora Server 44 on default kernel version 7.1 with Docker version 29.6 with default settings. These were the latest available software versions at the time of the experiments.

[7]https://github.com/rubensworks/jbr.js/tree/4a1e85f

connected by a virtual network, with 6 CPU threads and 8GB memory dedicated for the query engine, and 2 CPU threads and 4GB memory dedicated for the Linked Data file server.

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 [21], 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 replication count for individual query executions was set to 6. The experiments and our results are available online[8] for reproducibility and validation.

6. Results

This section details the metrics we use for analysis, and presents the analysis itself. The full measurements are available online alongside the experiments.

6.1. Queries considered in the analysis

Even with the complex queries excluded from the benchmark setup, the baseline experiments failed to execute all queries successfully. This can be seen in Table 2. Throughout the remainder of the results section, we only consider the 46 queries that executed successfully across all experiment test cases, ensuring the measurements are directly comparable.

6.2. Result metrics

Within this work, our focus is on *query performance*, where this performance can be measured from various different perspectives:

1. The *time to first solution mapping*, which is most visible to users. The earlier the engine produces the first solution mapping, the more responsive it may appear.
2. The *diefficiency* of the execution, that is, how the solution mappings are produced over the duration of the execution. This is detailed below.
3. The *time to last solution mapping* and the total *duration*. The earlier a query produces its last solution mapping and completes the execution, the faster it may appear.

The *diefficiency metrics* [49] 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 solution mappings with different rates over time. 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 solution mappings, where lower is better. The metrics are calculated as an integral of the solution mapping distribution function over time, illustrated in Fig. 2.

Within this work, we chose to employ *dieff@k*, and assign k to the total number of solution mappings for a given query. This allows us to aggregate the measurements across queries producing different numbers of solution mappings, while still maintaining a connection between the metric and the continuous efficiency represented by it.

For each of these metrics, we calculated the margin of error across the replications, assuming standard deviations, using a 95% confidence interval.

[8] https://github.com/surilindur/comunica-experiments/tree/424aeb2

Table 2
Overview of the benchmark experiment test cases, including the total runtime of all the queries, including replications. Even the baseline configurations failed to successfully execute all queries.

Experiment case	Successful queries	Total HTTP requests	Total solution mappings
baseline-traversal-heuristics	50 / 75	11,083	1,176
baseline-traversal-heuristics-0ms	48 / 75	3,710	1,174
baseline-traversal-heuristics-100ms	49 / 75	3,710	1,174
baseline-variable-counting	49 / 75	11,082	1,176
baseline-variable-counting-0ms	50 / 75	3,703	1,174
baseline-variable-counting-100ms	49 / 75	3,707	1,174
baseline-void-cardinality-hagedorn	48 / 75	11,086	1,176
baseline-void-cardinality-hagedorn-0ms	48 / 75	3,712	1,174
baseline-void-cardinality-hagedorn-100ms	48 / 75	3,473	971
baseline-void-cardinality-predicate	49 / 75	11,085	1,176
baseline-void-cardinality-predicate-0ms	47 / 75	3,710	1,174
baseline-void-cardinality-predicate-100ms	48 / 75	3,708	1,174
restart-polling-100ms-hagedorn-once	48 / 75	3,713	1,174
restart-polling-100ms-hagedorn-unlimited	48 / 75	3,714	1,174
restart-polling-100ms-predicate-once	47 / 75	3,713	1,174
restart-polling-100ms-predicate-unlimited	48 / 75	3,714	1,174
restart-polling-1000ms-hagedorn-once	48 / 75	3,712	1,174
restart-polling-1000ms-hagedorn-unlimited	47 / 75	3,713	1,174
restart-polling-1000ms-predicate-once	48 / 75	3,712	1,174
restart-polling-1000ms-predicate-unlimited	46 / 75	3,713	1,174
restart-reactive-hagedorn-once	48 / 75	3,713	1,174
restart-reactive-hagedorn-unlimited	47 / 75	3,714	1,174
restart-reactive-predicate-once	47 / 75	3,713	1,174
restart-reactive-predicate-unlimited	47 / 75	3,712	1,174

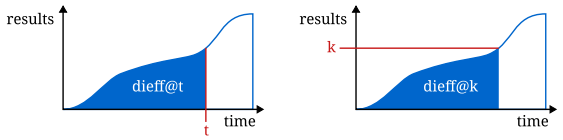

Fig. 2. The *diefficiency metrics* are calculated as an integral of solution mapping arrivals over time. For diefficiency at time t, higher is considered better, as the engine has produced more solution mappings within that given time. For diefficiency at k solution mappings, lower is considered better, as the engine has spent less time producing the given number of solution mappings.

6.3. Execution time results

The total query execution times for different experiment cases are plotted in Fig. 3, from which one can observe that rate-limiting increases total execution time by an average of 77%. Introducing 100 milliseconds of latency increases the execution times to an average of 4.4× that of the no-rate-limiting and no-network-latency scenario. Applying restart-based techniques with rate-limiting active yields an average 2% reduction in execution time compared with no restarts, ranging from a 9% average reduction to a 3% increase depending on the chosen method. Introducing 100 milliseconds of latency raises this to a 4% reduction on average, ranging from a 10% average reduction to a 1% increase depending on the method. With an average margin of error of 18.7% for query duration, these values fall within the error margins.

The time to first solution mapping, illustrated in Fig. 4, follows a slightly different but still similar trend, with rate-limiting yielding the first solution mapping 4% earlier on average, while introducing 100 milliseconds of latency causes the first solution mapping to take around 5.5× as long as without rate-limiting or added latency. Applying

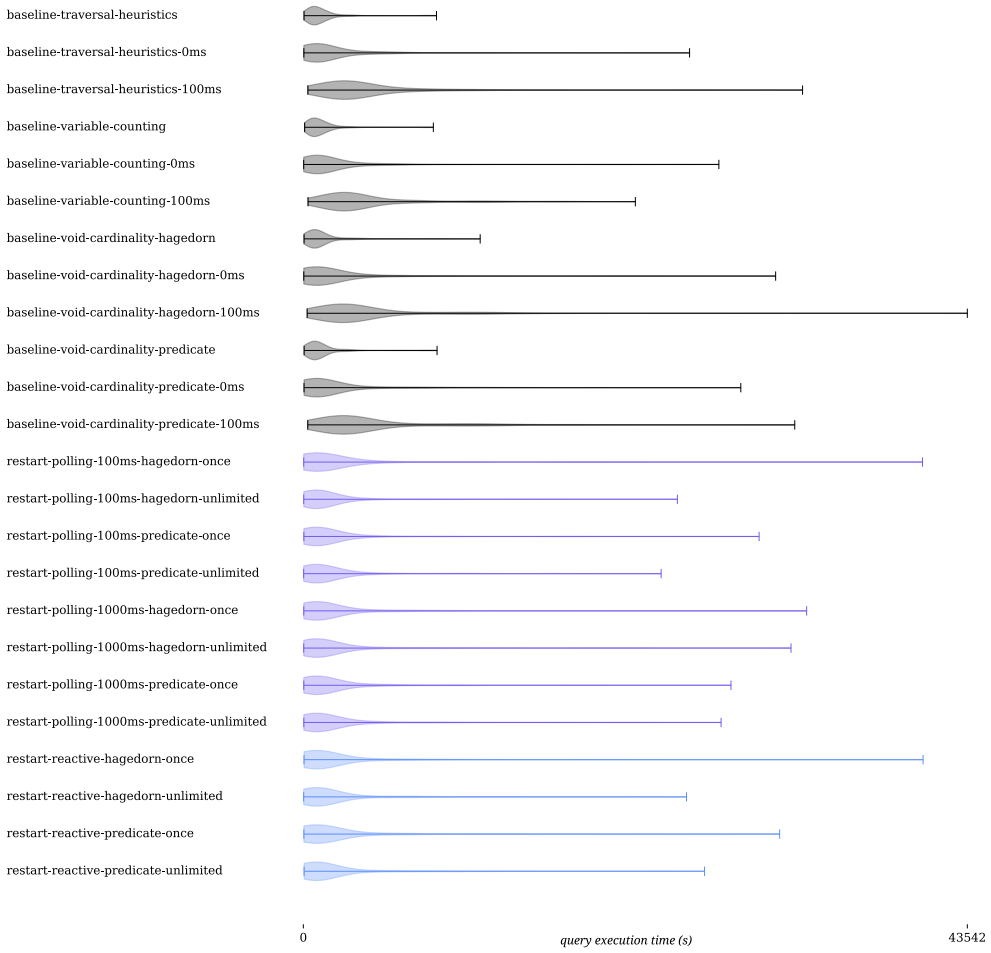

Fig. 3. The distribution of *individual query execution times* by configuration. Rate-limiting substantially increases the times, and network latency increases them further.

restart-based techniques with rate-limiting active produces the first solution mapping an average of 4% later than without restarts, ranging from 1% to 6% later depending on the method. Introducing 100 milliseconds of latency reduces this to 2% on average, with a range of 0% to 5% depending on the method. With an average margin of error of 19.3% for time to first solution mapping, these values fall within the error margins.

The time to last solution mapping, illustrated in Fig. 5, exhibits similar behaviour to that of the time to first solution mapping, with a 9% average increase when rate-limiting is added, and a 5.4× increase with 100 milliseconds of added network latency. Applying restart-based techniques with rate-limiting active produces the last solution mapping an average of 3% later than without restarts, ranging from 1% to 6% depending on the method. Introducing 100 milliseconds of latency reduces this to 2% on average, with a range of 1% to 5% depending on the method. With an average margin of error of 17.9% for the time to last solution mapping, these values fall within the error margins.

6.4. *Diefficiency metric results*

The *dieff@full* metrics for different experiment cases are plotted in Fig. 6, from which one can observe that rate-limiting increases total diefficiency by an average of 62%, with increases ranging from 55% to 69%. With lower diefficiency being better, this represents a reduction in performance. Introducing 100 milliseconds of latency increases diefficiency to 6.6× the original, which is consistent with the execution time increases. Applying restart-

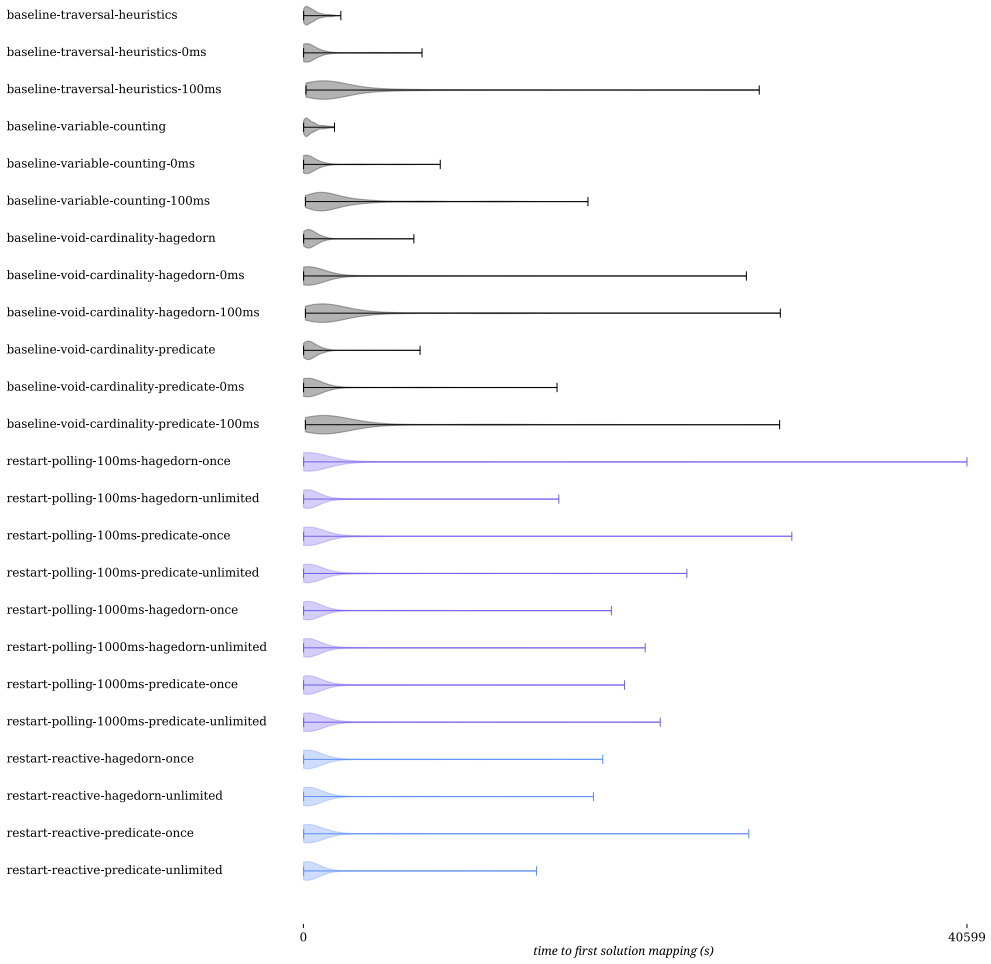

Fig. 4. The distribution of *time to first solution mapping* by configuration, for baseline, as well as interval-based and reactive evaluations.

based techniques with rate-limiting active yields 4% higher diefficiency on average than without restarts, ranging from 2% to 7% depending on the method. Introducing 100 milliseconds of latency reduces this to 3% on average, with a range of 1% to 5%.

Overall, applying restart-based techniques results in around 1%–2% execution time reductions on average, while delaying the first solution mapping by 2%–4% and the last by 2%–3% on average compared with not using restarts, depending on the network latency. Diefficiency also increases by around 3%–4% on average depending on the network latency. With the average margin of error in the range of 18%, however, all these measurements fall within error margins.

6.5. Resource consumption results

The resource consumption for entire experiment test case executions is illustrated in Fig. 7. The benchmark runner tool *jbr.js* collects these metrics through the Docker API separately from query execution, making it impossible to accurately attribute resource consumption to individual query executions. Thus, the resource consumption cannot be associated with any specific query. From the illustration, however, one can see that the baseline approaches without rate-limiting or added network latency consume processor and memory resources most intensively. The different experiment cases use system resources to roughly the same extent overall, with rate-limited cases distributing this usage over a longer period.

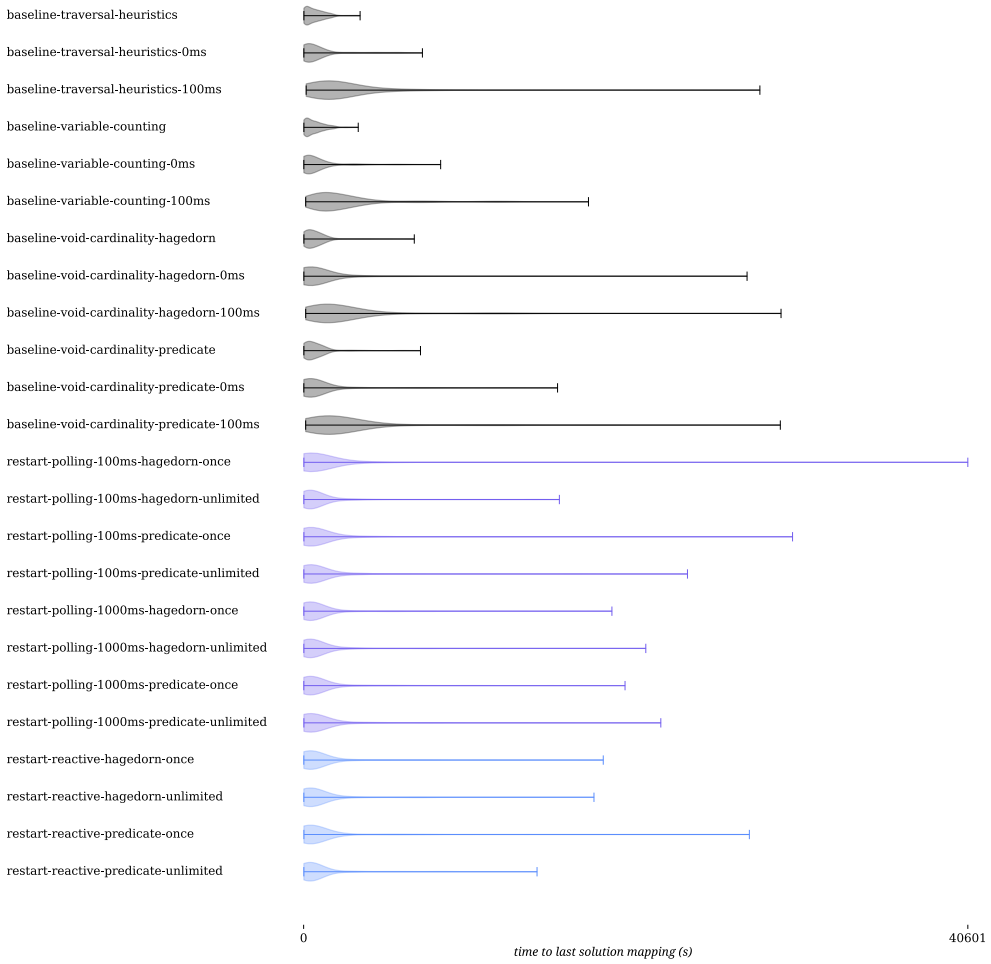

Fig. 5. The distribution of *time to last solution mapping* by configuration, for baseline, as well as interval-based and reactive evaluations.

7. Discussion

Although there were reductions in average query execution times attainable through the use of restart-based techniques, these reductions fall within the margin of error of the experiments. Likewise, the increases in diefficiency, as well as time to first and last solution mapping output, fell within the margin of error. These results appear to confirm the conclusions of prior work by Hartig and Özsu [17], in that client-side optimisations are unlikely to result in measurable overall query performance improvements within link traversal contexts. Specifically, when rate-limiting and network latency are considered, this appears to hold true even in environments with well-defined structure to assist in data discovery, such as the Solid initiative [1].

Although prior work has shown client-side techniques to provide measurable improvements, ranging from practical 36% reductions in execution time [11] to theoretical 50% reductions through the use of an oracle [21], such improvements appear unattainable under more realistic network conditions. When the execution time is increased by anywhere from 1.7× to 4.4×, and the majority of the increase is spent on data access, these changes in local processing become much less significant.

Thus, we reject Hypothesis 1 because restart-based approaches did not provide any measurable improvements in environments with rate-limiting and network latency. With the experiment measurements all falling inside the margin of error, we are also forced to reject Hypothesis 2 under the settings of our experiments, due to no difference between the join plan evaluation approaches. Likewise, we must also reject Hypothesis 3 due to lack of measurable

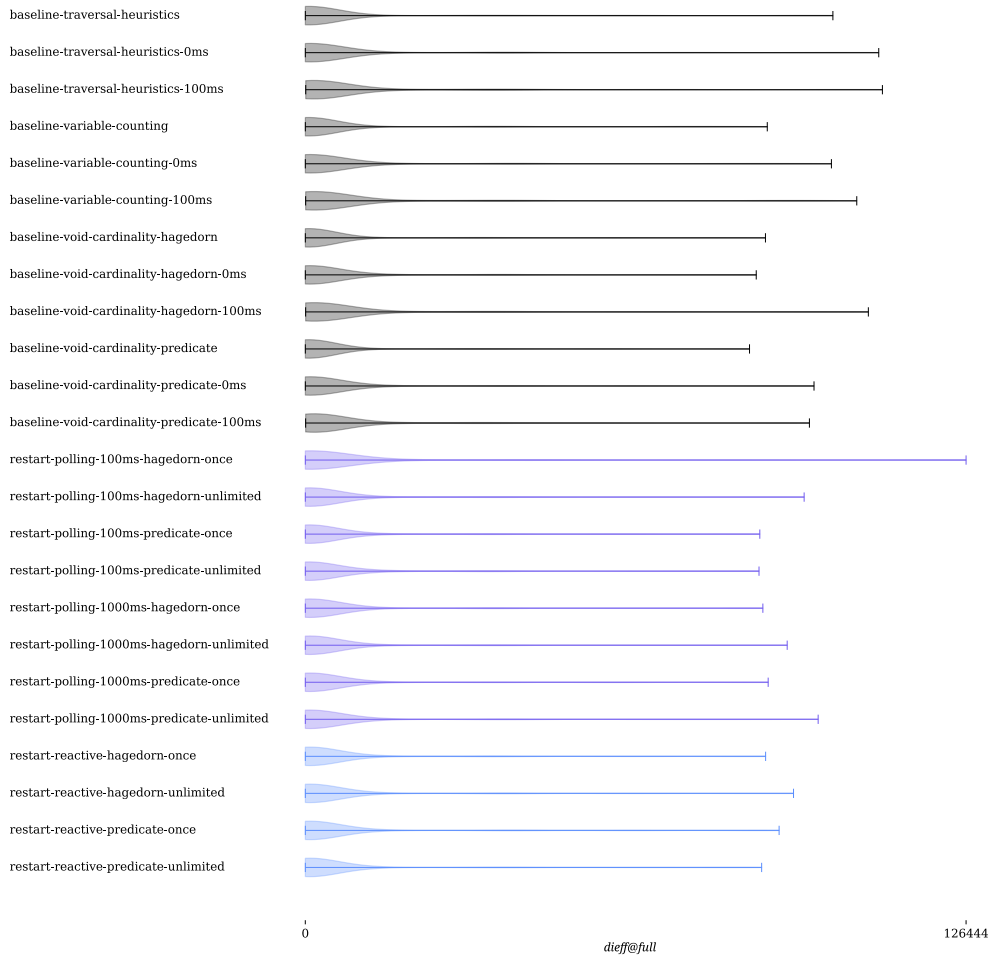

Fig. 6. The distribution of *dieff@k* by configuration, for baseline, as well as interval-based and reactive evaluations, with *k* set to the total number of solution mappings.

difference between single and multiple plan restarts, and Hypothesis 4 must also be rejected due to a lack of measurable difference between the cardinality estimation techniques. This applies within the context of our work, but may not hold outside it, such as under less extreme network overhead.

8. Conclusions

The results from our experiments lead us to conclude that network latency and rate limiting can entirely negate any improvements attainable through client-side optimisations. Even though prior work has demonstrated measurable improvements through client-side techniques [11, 21], under conditions with rate limiting and realistic levels of network latency, these improvements are greatly diminished and fall within the margin of error, in line with the conclusions of Hartig and Özsu [17]. This is our answer to Question 1: client-side techniques will only have a marginal impact on overall query performance, even in well-structured traversal environments such as Solid, when operating under more realistic levels of network overhead.

The primary takeaway from our work is therefore that network latency or rate limits should always be taken into consideration in future work and benchmarks, as these have the potential to significantly alter the outcome of any such work, and impact its application in real-world settings. This does not mean client-side techniques cannot have

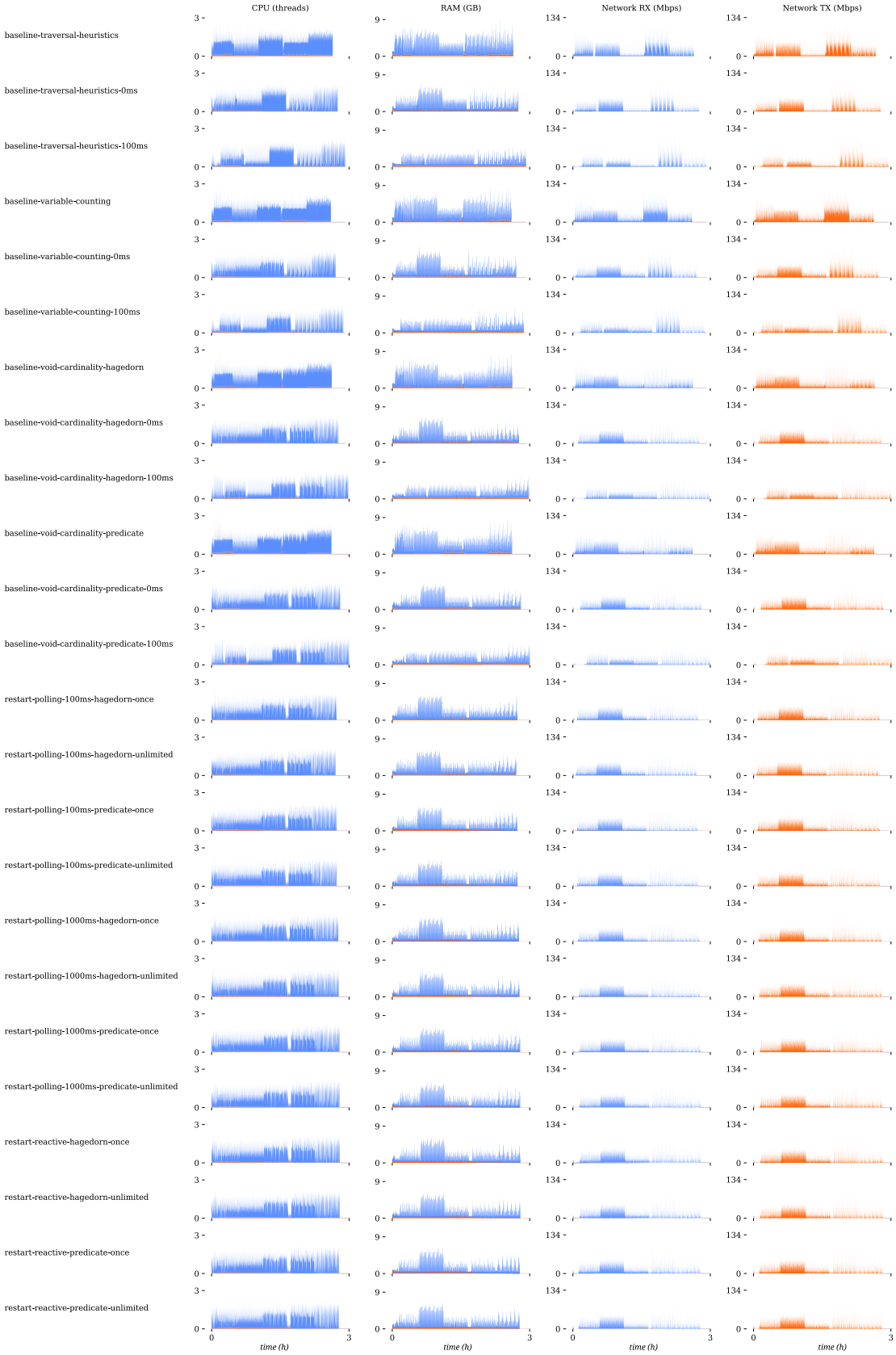

Fig. 7. The resource consumption of different experiment test cases over the duration of the entire experiment run, for the query engine and the Solid server. The *jbr.js* tool uses Docker to collect these metrics, and it does so separately from the individual query executions, making it technically impossible to accurately split the resource consumption by query.

a significant impact, but rather that we should consider more realistic scenarios, instead of zero-latency, unrestricted-rate, ideal laboratory settings.

We also believe future work on link traversal optimisation should consider ways to reduce the amount of network traffic, or to circumvent the overhead imposed by data access over the network. Avoiding network overhead should allow local query optimisation techniques to demonstrate their effectiveness. For practical link traversal to support the adoption of decentralisation initiatives, reducing mandatory network overhead offers considerable optimisation potential.

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] T. Berners-Lee and K. O'Hara, The read–write linked data web, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* **371**(1987) (2013), 20120513.
[3] C. Lemmer-Webber, J. Tallon, E. Shepherd, A. Guy and E. Prodromou, ActivityPub, W3C Recommendation, W3C, 2018, https://www.w3.org/TR/2018/REC-activitypub-20180123/.
[4] R. Cyganiak, D. Wood and M. Lanthaler, RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation, W3C, 2014, https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.
[5] M. Zignani, S. Gaito and G.P. Rossi, Follow the "mastodon": Structure and evolution of a decentralized online social network, in: *Proceedings of the International AAAI Conference on Web and Social Media*, Vol. 12, 2018, pp. 541–550.
[6] L. Vandercruysse, S. Van Damme and R. D'Hauwers, SolidLab Vlaanderen deliverable 8.4 b-Rapport Solid readiness: Drijfveren & hindernissen voor de publieke sector (2023).
[7] 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/.
[8] J. Wright and E. Bremer, Linked Web Storage Protocol 1.0, W3C Community Group Working Draft, W3C, 2026, https://www.w3.org/TR/2026/WD-lws10-core-20260728/.
[9] R. Taelman, Towards Applications on the Decentralized Web using Hypermedia-driven Query Engines, *ACM SIGWEB Newsletter* (2024).
[10] R. Taelman, From Server-centric to Client-centric Data Integration over Decentralized Knowledge Graphs with Personal Query Engines, in: *Joint Proceedings of Posters, Demos, Blue Sky, Workshops, and Tutorials of the 22nd International Conference on Semantic Systems*, 2026.
[11] 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.
[12] 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.
[13] T. Berners-Lee, Linked Data - Design Issues, 2006. http://www.w3.org/DesignIssues/LinkedData.html.
[14] O. Hartig and A. Langegger, A database perspective on consuming linked data on the web, *Datenbank-Spektrum* **10** (2010), 57–66.
[15] O. Hartig, SPARQL for a web of linked data: semantics and computability, in: *Proceedings of the 9th International Conference on The Semantic Web: Research and Applications*, 2012, pp. 8–23. ISBN 9783642302831.
[16] G. Halasz Frank, Reflections on NoteCards: Seven issues for the next generation of hypermedia systems, *Communications of the ACM* **31**(7) (1988), 836–852.
[17] 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).
[18] 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.
[19] L. Heling and M. Acosta, Robust query processing for linked data fragments, *Semantic Web* **13**(4) (2022), 623–657.
[20] 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.
[21] R. Taelman and R. Verborgh, Link Traversal Query Processing Over Decentralized Environments with Structural Assumptions, in: *International Semantic Web Conference*, 2023, pp. 3–22.

[22] A. Deshpande, Z. Ives, V. Raman et al., Adaptive query processing, *Foundations and Trends in Databases* **1**(1) (2007), 1–140.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] 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.
[31] 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.
[32] 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.
[33] 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.
[34] B.-E. Tam, R. Taelman, P. Colpaert and R. Verborgh, Opportunities for Shape-based Optimization of Link Traversal Queries, in: *Proceedings of the 16th Alberto Mendelzon International Workshop on Foundations of Data Management*, 2024.
[35] 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.
[36] 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.
[37] 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.
[38] 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/.
[39] 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.
[40] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C Recommendation, W3C, 2013, https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.
[41] T. Urhan and M.J. Franklin, Xjoin: A reactively-scheduled pipelined join operator, *Bulletin of the Technical Committee on* (2000), 27.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] C. Pelsser, L. Cittadini, S. Vissicchio and R. Bush, From Paris to Tokyo: On the suitability of ping to measure latency, in: *Proceedings of the 2013 conference on Internet measurement conference*, 2013, pp. 427–432.
[47] J. Van Herwegen and R. Verborgh, The Community Solid Server: Supporting research & development in an evolving ecosystem, *Semantic Web* **15**(6) (2024), 2597–2611.
[48] V. Balseiro, J. Zucker and S. Capadisli, Solid WebID Profile, W3C Community Group Draft Report, W3C, 2026, https://solid.github.io/webid-profile/.
[49] 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.