Incremunica: Web-based Incremental View Maintenance for SPARQL

Maarten Vandenbrande[1][0009−0003−6470−9123], Ruben Taelman[1][0000−0001−5118−256X], Pieter Bonte[2][0000−0002−8931−8343], and Femke Ongenae[1][0000−0003−2529−5477]

[1] IDLab, Ghent University – imec, Ghent, Belgium, firstname.lastname@ugent.be
[2] Department of Computer Science, KU Leuven Campus Kulak, Belgium, pieter.bonte@kuleuven.be

Abstract. The dynamic nature of Linked Data from IoT devices, social media, and the financial sector requires efficient mechanisms to keep SPARQL query results up to date, as traditional reevaluation methods are computationally expensive and impractical. Incremental view maintenance (IVM) offers a more efficient alternative by updating query results incrementally. However, existing engines lack support for federated querying, dynamically adding and removing sources during query execution, SPARQL Query Language support, multiple IVM techniques, and client-side execution. In this paper, we present Incremunica, an incremental query engine that addresses these gaps. Incremunica uniquely integrates multiple state-of-the-art incremental operators, allowing it to adapt to different queries and data for optimal performance. In this article, we provide 1) a requirements analysis comparing Incremunica to related work, 2) an explanation of Incremunica's architecture and features, 3) a performance evaluation showing improvements over reevaluation, and 4) a demonstration of its benefits through a social media watch party application.
Resource type: software framework
License: MIT
DOI: https://doi.org/10.5281/zenodo.15085224
URL: https://github.com/comunica/incremunica

Keywords: Incremental View Maintenance · SPARQL · Dynamic Linked Data

1 Introduction

Dynamic data refers to the additions and deletions of datasets, as well as additions, deletions, and updates of the data points within them. It emerges from a wide variety of sources across multiple domains. Sensors and IoT devices are prominent contributors, generating continuous streams of data in smart homes, industrial environments, and healthcare applications, such as through wearable devices [9,26,40]. Social media platforms (e.g., Bluesky and Mastodon), e-commerce systems (e.g., Amazon), and collaborative document platforms (e.g., overleaf), also produce dynamic data in real-time, such as posts, likes, and user interactions [34]. In the financial sector, systems for fraud detection, stock exchanges, and banking operations rely heavily on the analysis of dynamic data [27]. Finally, dashboards for network monitoring, IT systems, and business operations require fast updates to maintain a smooth operation [12,30,27,10]. For

all these domains, it has been shown that semantics play a crucial role in interpreting, integrating, and deriving meaningful insights from diverse and dynamic data sources, e.g., to realize optimized patient monitoring [18], building management [37], network management [48], predictive maintenance [43], and Fintech [53]. By using semantic technologies, systems can represent the data with rich context, enabling interoperability between heterogeneous sources, enhancing query capabilities, and improving machine readability and AI integration.

Maintaining query results over dynamic data, called incremental view maintenance (IVM), is challenging as recomputing the complete query results every time the data changes is not always efficient or fast enough. Incremental query engines are crucial to deal with this challenge. They use the changes in the dynamic data to calculate changes in the query result. A few incremental SPARQL query engines for the Semantic Web exist, such as SEPA [40], INSTANS [39], Diamond [35], IncQuery-D [44], and AWETO [38]. However, they each implement only one algorithm to enable incremental SPARQL operators to incrementally update the results, which limits their overall performance potential. Ensuring high performance is especially critical in environments with high-velocity data streams, such as those generated by sensors and IoT devices, as well as in applications where low latency is paramount, like dashboards and financial systems, where timely updates are essential for effective decision-making and smooth operation. Moreover, most of the current engines lack support for federated querying, which is crucial when querying the Semantic Web, where data aggregations of heterogeneous sources is a common requirement. Additionally, the ability to dynamically add and remove sources during query execution is equally important in such environments, as new sources may become relevant or interesting to query, especially in decentralized systems like social media platforms. Additionally, many engines lack full SPARQL Query Language support, which is essential for the usability of the engine. Furthermore, none offer client-side execution, a feature that enables privacy-preserving applications and reduces latency in use cases such as social media, dashboards, and document-sharing platforms.

In this paper, we introduce Incremunica, an incremental query engine built as an adaptation/extension of Comunica [45], a modular and easily extendable SPARQL query engine for the Web. The main contributions of this resource are providing **1.** an incremental query engine for SPARQL, supporting both additions and deletions, built using Web technologies. **2.** Incremunica is able to run federated queries over decentralized sources, that offer different Web interfaces to access their data. Furthermore, the ability to use different Web interfaces allows Incremunica to pose as a benchmarking platform to evaluate existing, and future Web interfaces from an IVM's perspective. **3.** For fast incremental evaluation, Incremunica supports multiple versions of incremental operators in one engine, allowing it to select the ideal incremental operator for the use case.

The structure of this article is as follows. The next section first gives a positioning of the related work, which is then followed by the requirements under which we have built Incremunica. Next, in Section 3, the architecture and design decisions are laid out. Section 4 gives a performance evaluation and comparison compared to reevaluation. Following this, we give a use case in the realm of a social media platform, explaining how our watch party with decentralized storages is implemented with Incremunica. Finally, we end the article with a conclusion and roadmap.

2 Related Work & Requirements

This section first describes the related work on incremental query engines for the Semantic Web. Then, requirements are derived for our incremental engine, Incremunica, based on these works and our specific goals. We then evaluate the related work to determine how well they fulfill these requirements, and thus highlight the need for Incremunica.

2.1 Related Work

First, incremental query engines are used to improve the performance of sliding windows in RDF Stream Processing [49]/Streaming Linked Data [14]. A sliding window is a technique used to process and analyze streaming data by continuously maintaining a fixed-size or time-bounded subset of the data. This window moves incrementally as new data arrives, ensuring that only the most recent elements within the defined bounds are considered for processing. In this context, incremental operators can be optimized beforehand because the boundaries of the sliding window determine when an element (or query result) will go out of scope [33,13]. For this reason, these incremental query engines are not considered in our requirement analysis.

Incremental query engines are also valuable in complex event detection, where a stream processing system evaluates and monitors a system's current conditions or state based on streaming data. Incremental query engines can efficiently query over these changing states instead of reevaluating the query each time the state changes [27]. An example of such a complex event detection engine for the Semantic Web is INSTANS [39]. It is a Rete-based incremental query engine, that claims triple addition and deletion support for network, triple stores, files or other processes. It also uniquely has a pipeline to update its internal RDF store based on the query results.

A big adopter of incremental query engines in the Semantic Web is Link-Traversal-Based Query Processing (LTBQP) [24,41,25], more specifically, where the query engine discovers additional sources during query execution. The data from these sources are then incrementally added to the query engine to increase the recall of the query results. However, LTBQP often only considers additions of sources and data [45,23,32]. Therefore, we will not consider them when evaluating the requirements. Only Diamond [35] uses the Rete algorithm [20] to incrementally evaluate the query. Diamond only supports additions of triples and sources, but is able to use deletions internally to remove the monotonicity of the *OPTIONAL* and *MINUS* operator in SPARQL. The idea of removing monotonicity is further explained in Section 3.1. Diamond has support for a limited subset of the SPARQL Query Language: Basic Graph Patterns (BGP), *MINUS*, *OPTIONAL*, *UNION*, and *FILTER*. Finally, it only supports fetching RDF files hosted over an HTTP Web interface.

Lastly, in the field of database systems, incremental query engines are found to be crucial for building incremental view maintenance (IVM) systems that cache frequently asked queries, especially when aggregating the results from different decentralized data sources [16,21]. The concept of IVM has also been adapted to the Semantic Web, for example, SEPA [40] is an engine built to improve the performance of publish/subscribe architectures of Internet of Things (IoT) networks. Clients can subscribe to queries that are

evaluated by the SEPA engine on data streams that are pushed to the engine by IoT sensors. The SEPA engine improved upon the work of Smart-M3 [36] that used reevaluation to update the query result by using the counting algorithm from Gupta et al. [21]. Next, in the same field of publish/subscribe architectures, the engine from Abdullah et al. [9] increased the performance of incremental query evaluation by using the Rete algorithm. The last fully incremental query engine that supports both additions and deletions is IncQuery-D [44]. This engine was built to find inconsistencies/resolve constraints in complex models to aid in model-driven software engineering. IncQuery-D uses the fact that Rete-networks can be distributed over multiple computational nodes to parallelize the incremental evaluation. Finally, AWETO [38] is again a Rete-based incremental query engine that only supports incremental evaluation of bulk insertions to their local storage. As AWETO doesn't support deletions, it will also be excluded from the requirement analysis.

2.2 Requirements

Based on the related work and our goals for Incremunica, we have defined a set of requirements. We will outline and explain these requirements in detail, and then summarize how the related work, and Incremunica meets or falls short of these requirements.

The most basic requirement for an incremental query engine is to have support for both additions and deletions of data (R_1). Most engines in the related work can perform IVM over either a locally maintained RDF store or access the data through a single interface. These interfaces are either local, like a file or another process, or are available on the Web through a Web interface like a SPARQL endpoint or a file interface over HTTP. A query engine for the Semantic Web should be able to query over multiple heterogeneous interfaces (R_2), requiring the need to support federated query techniques. Incremental query engines that support multiple Web interfaces can allow for more efficient view maintenance. Furthermore, they should also be able to alter the set of sources (add and remove) during query execution (R_3), to dynamically limit the data over which it queries.

Many of the current incremental query engines realize IVM by using the counting algorithm, where the query is materialized based on an addition or deletion and then evaluated with a non-incremental engine. Although this approach is easy to implement, and more performant than re-evaluation in most cases [29,40], it is not the most performant technique. Two alternative techniques for achieving IVM with better performance are Rete-based methods [20] and higher-order delta queries, also known as higher-order IVM [28,10]. Both approaches have advantages and disadvantages in certain scenarios, supporting both enables the selection of the most efficient approach for optimal performance (R_4). Further explanations of these techniques are found in Section 3.1.

Currently, there are no incremental query engines available for client-side evaluation (R_5). Using Web technologies allows an engine to be used client and server side (with for example Node.js). Finally, it is important to support the complete SPARQL query language for SELECT queries (R_6), including but not limited to property paths, aggregations, and ordering. This is more thoroughly explained in Section 3.1. Below, a summary of the requirements is given for readability.

R_1 Supports additions and deletions
R_2 Allows to support federated querying
R_3 Supports adding and removing sources
R_4 Fast incremental operators
R_5 Uses Web technologies
R_6 Supports all SPARQL SELECT operators

	R_1	R_2	R_3	R_4	R_5	R_6
Diamond	✓	✓(1)		✓(2)		
SEPA	✓					✓
Abdullah et al.	✓			✓(2)		
INSTANS	✓	✓(1)		✓(2)		✓(3)
IncQuery-D	✓			✓(2)		(4)
Incremunica	✓	✓	✓	✓	✓	✓(3)

Table 1. Requirements of an incremental query engine. **1.)** Only supports triple interfaces. **2.)** Only supports Rete-based incremental operators. **3.)** No support for repeating property paths. **4.)** Could not find mention on completeness of SPARQL operator support.

Table 1 shows how the related work meets the requirements we described above. This analysis highlights the limitations of existing incremental query engines in meeting the full range of requirements for Semantic Web applications. While engines such as Diamond, and INSTANS provide partial support, they fall short in areas like dynamically adding and removing sources, the use of Web technologies, support for higher-order delta queries, and full SPARQL SELECT support.

3 Incremunica

Our resource contribution is called Incremunica. The source code for Incremunica is available from Github at `https://github.com/comunica/incremunica` under an MIT license (canonical citation: https://doi.org/10.5281/zenodo.15085224). The query engine is an adaptation/extension of Comunica [45], a query engine written in TypeScript. Incremunica is organized as a monorepo and leverages dependency injection, enabling easy extensibility and modularity. This structure allows for comprehensive testing, both with unit testing on individual packages and end-to-end testing, resulting in 100% test coverage. To ensure high code quality and maintainability, Incremunica adheres to standard practices, including linters for code analysis and semantic versioning for version management.

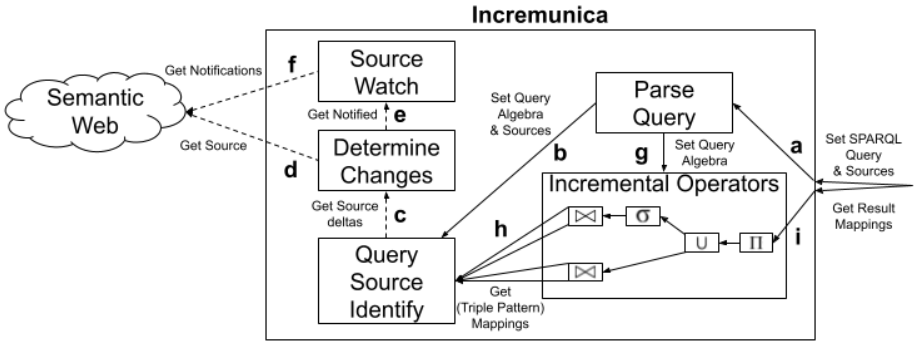

Fig. 1. High-level overview of the Incremunica architecture.

Figure 1 presents a high-level overview of Incremunica's architecture. When a query and its associated sources are passed to Incremunica (**a**), the query is parsed by the default Comunica components into a SPARQL algebra. Next, the sources and the SPARQL algebra are sent to the *Query Source Identify* component (**b**). This component identifies the type of the source, such as an RDF store or a Web interface like an RDF file hosted over HTTP. In case of an RDF store, the *Incremental Operators* setup by the query algebra (**g**) can immediately get a change stream of mappings from the RDF store in the *Query Source Identify* component. Internally, the *Query Source Identify* matches the triple patterns of the query to the store. Note that the RDF store needs to support subscribing to changes of a triple pattern. This means if a triple is added that matches a triple pattern, the mappings need to be sent in the change stream of that triple pattern. Finally, The pipeline of *Incremental Operators* will return a stream of result mappings to the SPARQL query (**i**). Where each result mapping has a boolean context entry, "*isAddition*", that specifies if the mapping is added or deleted from the result set. If instead of an RDF store, a URI is passed as the source to the *Query Source Identify* component (**b**), then the *Query Source Identify* component will identify the Web interface of that URI. Based on this, the *Query Source Identify* component will get the changes, i.e., deltas, of the source from the *Determine Changes* component (**c**). If the source is an RDF file, the *Determine Changes* component will fetch the RDF triples (**d**) and index them into an RDF store. It will then be responsible for keeping track of future changes to that RDF file and updating the RDF store. Finally, this store is shared with the *Query Source Identify* component (**c**) where triple patterns are matched to the store. Some Web interfaces can handle triple patterns, basic graph patterns (BGP), or full queries. In this case, a subset or the entire query can be offloaded to the source if that part of the query is evaluated over only one source (**d**). The *Determine Changes* component will then return the deltas to that part of the query (**c**), which is then passed along to the *Incremental Operators* (**h**) or in case the source executes the entire query, it is passed to the user (**i**). If the query is evaluated over multiple sources with the capability to handle triple patterns or more, Incremunica fetches the individual triple patterns of the query from each source, and processes the rest locally (**h**) with the incremental operators. This is to guarantee complete answers to the query over multiple sources. Some Web interfaces, for example

hosting RDF files over HTTP, do not allow for having a stream of deltas from that file. In these cases, Incremunica also sets up a *Source Watch* component, which will notify the *Determine Changes* component on changes (**e**). Depending on the situation and the configuration of Incremunica, it will choose a suitable *Source Watch* technique, possibly by using a Web interface (**f**).

In the following sections, we will go more in depth on **1.** the *Incremental Operators* that calculate the changes in the query results from the changes in the data, **2.** the *Determine Changes* component, which determines which Web interfaces it can use to calculate the changes efficiently, and finally, **3.** the ways in which the *Source Watch* component can get notifications.

3.1 Incremental Operators

The simplest approach to achieving IVM is through the counting algorithm proposed by Gupta et al. [21]. This method materializes the query on updates and uses a non-incremental engine to evaluate the materialized query. However, Incremunica employs dedicated incremental operators that efficiently handle both additions and deletions. Deletions are propagated through the engine in the same way as additions, updating the operator's internal state. This ensures the state reflects a consistent query result as if the deleted data had never been added, guaranteeing complete and correct answers. The incremental operators used in Incremunica can be categorized based on how they handle state:

- **Linear operators:** These operators do not retain internal state and operate independently of previous data. Their behavior is identical to non-incremental counterparts. For instance, the *UNION* operator belongs to this category.
- **Stateful operators:** These operators maintain internal state for more complex operations like the *JOIN* operator. Two algorithms are implemented in Incremunica: Rete-based operators [20] and Higher-order IVM or higher-order delta query operators [28].

The first type of stateful operators are Rete-based operators. These operators take multiple inputs and perform a certain operation on them, for example, joining the two inputs based on the common variables. This approach is similar to symmetric joins [32,31], although this algorithm can only handle additions. A notable feature of the Rete algorithm is the ability to share intermediate results across multiple queries with overlapping operations. However, Incremunica does not implement this feature. The second stateful operators are the higher-order delta query operators. These operators are similar to the bind joins in a non-incremental engine [22], and are particularly effective when one input produces significantly fewer intermediate results than the other. Higher-order delta query operators achieve this by materializing the remaining query based on one input, similar to the counting algorithm. By removing one triple pattern during materialization, the resulting query becomes simpler. Unlike the counting algorithm, the resulting materialized query is then evaluated incrementally. Both types of stateful operators can be implemented using different indexing strategies. For example, indexing can enhance performance when matching overlapping variables, as seen in symmetric hash joins, or it can optimize deletions from the state. Although, the exact details are out of scope for this article.

Incremunica can also be of use in a Link Traversal setting, as it supports the addition and deletion of sources during execution time. The addition of sources can provide a more hands-on approach for developers to guided LTBQP [51], which allows the developer to use the result of a SPARQL query as input to the sources for another SPARQL query. For example, they could first query a dataset of academic publications to find the ones about IVM to then use the IRI's of these publications to get their abstracts. In an incremental setting, if new publications are added or removed from the dataset of academia publications, the results of the query for the abstract will reflect these changes dynamically. Also, the addition and deletion of sources during query execution includes support for changes in privacy. This means that during the maintenance of a query, a previously available source can become unavailable. In this case, all data from this resource will be removed from the internal state of Incremunica. Furthermore, as Incremunica is a full incremental query engine (supports deletions), it can also be used to solve the lack of monotonicity of some SPARQL operators, where a result can only be produced if all data has been processed. The lack of monotonicity can be an issue in Link Traversal, as it requires the engine to have the complete dataset before emitting results. Cheng et al. [15] evaluated the performance of a monotonic version of the *OPTIONAL* or left join operator, called the OPT+ operator. The OPT+ operator immediately emits a result where the results from the optional basic graph pattern (BGP) have not been merged. A downside to this technique is that the result set can be larger because of these premature results. In an incremental setting, it is possible to immediately emit the result like the OPT+ operator, but then remove this result if a mapping is found for the optional BGP. As previously mentioned, this technique has been shown by Diamond [35]. The *OPTIONAL* operator isn't the only operator that isn't monotonic, Incremunica also applies this technique to the *FILTER NOT EXIST*, *DIFF*, *MINUS*, *GROUP BY*, and *ORDER BY* operators.

A few SPARQL operators are currently not supported in Incremunica. While Incremunica does support property paths, it does not currently support repeating property paths. These are paths that use the '*' (zero or more) and '+' (one or more) operators to indicate repetition in a sequence of predicates. Lastly, Incremunica currently supports only SPARQL *SELECT* queries.

3.2 Determining Changes

The input of the incremental operators discussed in the previous section is the set of changes to the underlying data. This section discusses how Incremunica determines the changes or deltas in the data. This includes changes to both a local RDF store or online sources using Web interfaces, and outlines future directions for this research.

As mentioned in the explanation of the Incremunica architecture, Incremunica requires that local RDF stores give a delta stream of matches to a triple pattern. This can be achieved with any kind of traditional store (in-memory or on-disk). Incremunica has an in-memory implementation of this called the *Streaming Store*. As said, the *Streaming Store* will keep the stream open for not only the current matching triples, but also the triples that match the triple pattern in the future. This is accomplished by indexing both the triples and the triple patterns, so not only triple patterns can be matched to triples, but also added or deleted triples can be matched to the triple patterns.

In terms of Web based sources, only data dumps or RDF files hosted over HTTP are currently supported. These sources are dereferenced, parsed, and indexed into a *Streaming Store*. When a resource is updated, Incremunica takes a naive approach: it dereferences the new version, parses, and indexes it, and then calculates the changes by comparing the new and old *Streaming Store*. Blank nodes are usually equally skolemized, if not, the query result from the old blank node will be deleted and the query result from the new blank node will be added, ensuring complete and correct results. A possible future improvement would be to apply graph isomorphism algorithms for a more precise comparison. However, this may prove computationally expensive, making the current approach of deleting the results followed by adding it again more efficient in practice.

Future work will focus on supporting other existing interfaces to more efficiently calculate the changes in the data. The first candidate is Versioned Triple Pattern Fragments (VTPF) [46], which is an extension to the Triple Pattern Fragments (TPF) [52] interface. It allows to query the added and deleted triples that match a triple pattern between different versions of a resource. Another possible interface is the memento [42] protocol, which stores each version of the resource. This allows the query engine to purge the previous version of the document locally, slightly improving memory consumption. The third candidate is RDF Delta [6], which uses RDF PATCH logs that describe the changes to the RDF dataset. This allows to reconstruct the different versions. Finally, Linked Data Event Streams (LDES) [4] uses members to more efficiently store different versions of a resource. Each member is a collection of RDF triples with a similar shape. An addition, deletion, or update is stored immutably as an observation. As such, fetching all observations allows for the reconstruction of the dataset and its different versions. Other than actual interfaces, there also exist vocabularies that describe changes to an RDF dataset without actual interfaces, such as the ChangeSet vocabulary [3] and Activity Streams [1]. The next section will describe how these vocabularies could be used as an interface.

3.3 Source Watch

The previous sections explained how to get the changes in a resource and how to use them to maintain a materialized view incrementally. The remaining aspect to address is how Incremunica can be notified of changes to a resource. Notifications are typically handled through three main approaches: polling, pushing, and connection interfaces.

Polling a resource is the act of doing continuous HTTP requests to get the current state of a resource. Incremunica does simple HTTP HEAD requests to the server to find out if the HTTP entity tag (ETag) of the resource has changed. Doing a HTTP HEAD request decreases the bandwidth of polling as the complete resource does not have to be sent along, like with HTTP GET requests. The frequency of polling can be determined in advance by the user or by the cache control system of the resource. Polling can be useful for cache-controlled resources or if the latency of updates is not a major issue. It can also be useful for deferred maintenance [17], where the user defines when the materialized view needs to be updated, either time based or on request.

In a pushing approach, the data source notifies the query engine if changes to the data have occurred. This uses technologies like webhooks, HTTP POST, Server-Sent Events, etc. This leads to a lower latency than the polling approach. It has been shown by Van De Vyvere et al. [50] that in cases where 10 seconds or more is an acceptable latency, polling is a more scalable option compared to Server-Sent Events, due to the overhead of maintaining the connection. An example of an actual interface of this is Linked Data Notifications [5], where the data source sends the notifications to an inbox with HTTP POST requests.

Finally, a connection approach uses websockets to get notifications about the state of a resource. It allows for a duplex connection between the data source and the query engine, enabling the query engine to make alterations to the notification request, e.g., when the query engine is interested in an additional resource managed by the same notification service. An actual example of such an interface is the Solid Notification Protocol [7]. The current version of this protocol (0.2.0) uses websockets (or webhooks) and the Activity Streams [1] vocabulary to specify if a resource on a solid pod has changed. The full Activity Streams vocabulary could also be used in itself with a websocket interface to specify the exact changes (added and deleted triples) to a resource.

At the moment of writing, Incremunica supports deferred maintenance, polling, and the Solid Notifications Protocol, with plans of implementing all other Source Watches discussed in this section.

4 Performance Evaluation

This section discusses the performance of Incremunica with respect to handling updates, and also compares it against query reevaluation using Comunica. The benchmark can be accessed via the canonical citation: https://doi.org/10.5281/zenodo.14512790.

4.1 Evaluation set-up

For the evaluation, we used an adaptation of the LDBC Social Network Benchmark [19] called SolidBench [47]. SolidBench fragments the data from the LDBC Social Network Benchmark into different n-quads files, to mimic a network of decentralized data spaces. The LDBC Social Network Benchmark offers two workloads, short and complex. We selected the short workloads, as they provide a representative measure of Incremunica's performance while also being theoretically favorable to reevaluation due to their simplicity. These workloads describe updates to the data, consisting of insertions and deletions. For short queries 1 and 3, a friendship is added or removed according to insertion and deletion 8 in the specification [11]. For queries 2, and 4 to 7, a comment is added or removed as described by insertion and deletion 7. As Incremunica does not yet support recursive property paths (Section 3.1), queries involving the zero-or-more property paths (queries 2 and 6) were rewritten using an *OPTIONAL* clause to check for the existence of a single property.

Two experiments were conducted: offline and online evaluations. Both experiments follow the same methodology, differing only in the type of data sources. In the offline evaluation, data is stored and indexed in a single RDF store, with updates applied directly

to that store as described in the benchmark specification. The online evaluation uses the fragmented files as defined by SolidBench, accessed via HTTP GET requests. To benchmark Incremunica's update time, changes are applied to the files, and deferred evaluation (Section 3.3) triggers Incremunica to update the query results. Afterwards, Incremunica uses the naive way of determining changes described in Section 3.2. The online analysis represents the worst case for Incremunica, as it has to derive the changes between the files. Finally, to allow for a fair comparison and optimize the performance of reevaluation with Comunica, only the cache for the source that the benchmark had changed is invalidated.

The experiments were run on an Ubuntu 20.04.6 LTS x86_64 machine with Node.js v20.19.0 limited to an 80GB heap size, an Intel Xeon E5-2650 v2 @ 2.6GHz processor, and 129 GB of memory. Each scenario is run 30 times with a random person, each with a different number of friends, comments, posts, ect., returning a different number of results.

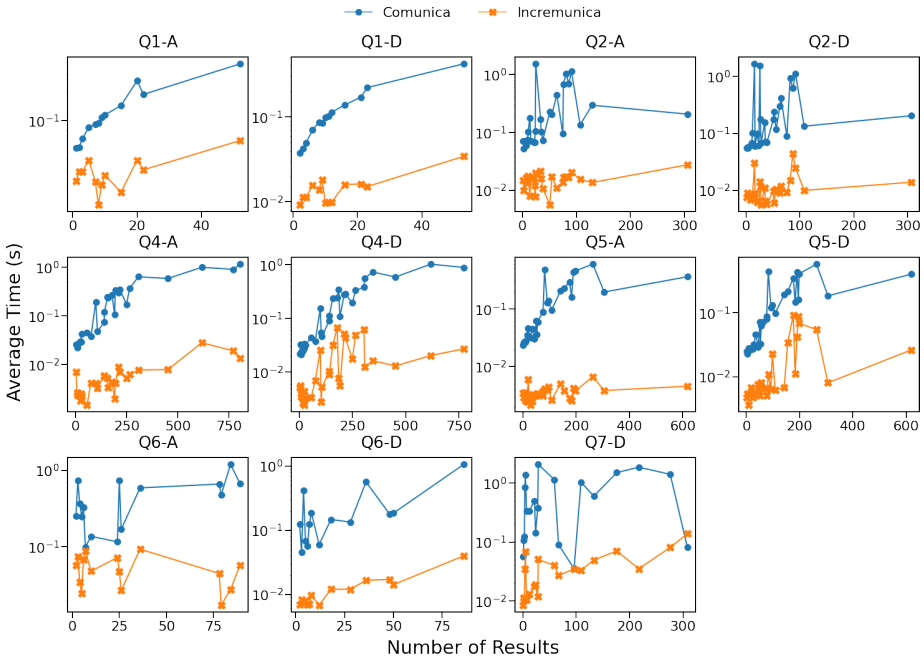

Fig. 2. Execution times for offline analysis, showing Incremunica outperforming reevaluation as query result size increases.

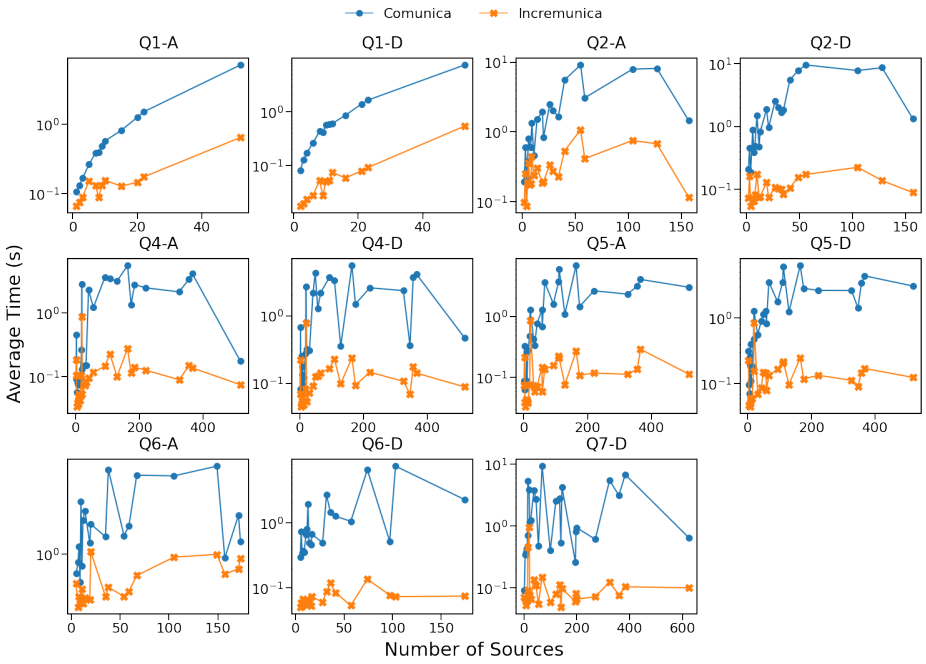

Fig. 3. Execution times for online analysis, highlighting Incremunica's advantage over reevaluation despite HTTP variability.

	Q1-A	Q1-D	Q2-A	Q2-D	Q4-A	Q4-D	Q5-A	Q5-D	Q6-A	Q6-D	Q7-D
min.	13.52	6.99	5.64	5.60	1.47	2.46	2.17	3.61	17.37	4.63	6.46
avg.	29.06	13.02	14.79	11.34	6.12	17.51	3.59	19.01	53.46	9.82	33.01
std. dev.	12.31	5.85	4.34	8.12	5.48	18.61	1.05	24.55	28.25	6.47	30.89
max.	62.24	34.38	27.46	44.45	27.95	66.94	6.68	92.30	114.49	39.76	137.75

Table 2. Minimum, average, standard deviation, and maximum evaluation times in milliseconds for Incremunica's offline analysis by query and update type.

4.2 Results & Discussion

The performance results are shown in Figure 2 and Figure 3. Each graph is annotated with the query number (workload number) and an "A" or "D" to indicate addition or deletion scenarios, respectively. Table 2 presents the minimum, average, standard deviation, and maximum evaluation times for the offline analysis, which provides a non-biased metric unaffected by HTTP latency or parsing speed. Scenarios Q3-A, Q3-D, and Q7-A never produced any results and are therefore excluded from the results. The offline analysis is presented as a function of the number of results the query produced, which aligns with the database size on which the query was evaluated. For the online analysis, the graphs are shown as a function of the number of sources, as HTTP requests are the main bottleneck in this context. This evaluation demonstrates that incremental query evaluation can significantly enhance performance when maintaining a materialized view in response to changing data.

Figure 2 shows the performance of the offline analysis. The results reveal that the execution time for reevaluation has a direct, increasing relationship with the number of results, while incremental view maintenance (IVM) often exhibits a constant relationship between execution time and the number of results. This agrees with the consensus of IVM that it is better in cases where the ratio between the size of the changes and the database is small. In all cases, IVM outperforms reevaluation, even for queries producing a small number of results, often by an order of magnitude.

In the online analysis, shown in Figure 3, a similar trend is observed, with Incremunica consistently outperforming reevaluation, again often by an order of magnitude. However, the online analysis is more variable than the offline one due to changes in the size of the fragmented files. This variability is unfavorable for the naive method of detecting changes discussed in Section 3.2, as it must identify differences between two larger sets of triples.

Finally, this evaluation also serves as a partial proof of the completeness and correctness of the operators used in Incremunica, as the results from the benchmark are constantly compared against the results from Comunica.

5 Use Case: A watch party with data spaces

We showcase the capabilities of Incremunica through the implementation of a specific use case from the field of data spaces like Solid [8], bluesky [2], and others. These initiatives aim to decentralize the Web by giving each person their own data vault in which their personal data is stored to which they can give permissioned access. This evolution creates many decentralized sources with heterogeneous Web interfaces. The data space of a person holds that person's online activity, creating large amounts of dynamic data. Redoing a query every time their data changes or the sources of the query are altered can become computationally expensive, slowing down applications, making them unusable.

A watch party is a combination of a video streaming service and chat. The idea is that a group of people can watch a synchronized video together on different clients, where the video stream on all clients is synced, and they can simultaneously chat about

their viewing experience. This kind of application requires the different clients to both sync up all messages and all video events like play, pause, and scrubbing. The user interface (UI) of our version of a watch party using Solid technologies can be consulted at: `https://github.com/SolidLabResearch/solid-watch-party`.

The creator of a watch party (the host) will make a resource available for all video controls and other watch party information, and store this information in their data vault. After the host invites other participants by sharing the URL, he can accept their invites in the application and gives them append permission on this resource. The participants will then create a resource as a message box, which holds all created messages by them, maintained in their own data vault. Each participant is required to give permission to all other participants (including the host) to read this message box resource. Figure 4 gives an overview of this architecture.

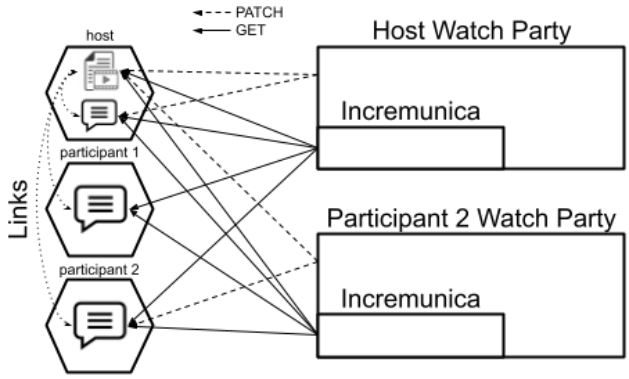

Fig. 4. An overview of the architecture for the watch party application.

Getting a constant stream of new messages for the watch party therefore requires us to perform an incremental query to the message boxes of all participants to keep up to date with these message boxes in case changes occur. Finally, it's possible that new users are added to the watch party whose messages and message box also need to be watched for changes. To achieve this, we can first query the message boxes by executing the following query on the room resource:

```
PREFIX schema: <http://schema.org/>
SELECT ?messageBoxes
WHERE {
  <[Room_IRI]> schema:subjectOf ?messageBoxes .
}
```

When performing this query with Incremunica, we get a constant stream of all message boxes now and in the future. Incremunica will set up a websocket connection with the host pod to ask for notifications if the room resource has changed. The result of this query can then be used as a list of sources for the following query:

```
PREFIX schema: <http://schema.org/>
SELECT ?message ?dateSent ?text ?sender
WHERE {
  ?message a schema:Message .
  ?message schema:dateSent ?dateSent .
  ?message schema:text ?text .
  ?message schema:sender ?sender .
}
```

This will query across the message box resources of the different participants pods and return a stream of added and deleted messages and their additional data. As mentioned in Section 3.3 a connection type interface allows for duplex communication. Incemunica will therefore use the existing websocket connection with the host pod to receive notifications for both the room resource and the message box resource of the host. For all incremental queries in the watch party application, the default version of Incremunica was used.

6 Conclusion and Roadmap

This work introduced Incremunica, an incremental query engine developed using Web technologies, aimed at addressing the need for efficient federated incremental querying over heterogeneous Web interfaces for both client-side and server-side applications. By uniquely integrating higher-order delta query-based and Rete-based incremental operators, Incremunica adapts to different queries and data for optimal performance. It also supports dynamic source addition and removal during query execution and supports most SPARQL SELECT operators, effectively filling gaps left by prior engines. Additionally, with its modular architecture and extensibility, Incremunica provides a platform to benchmark existing and future Web interfaces in the scope of IVM. Our performance evaluation demonstrated that Incremunica is often an order of magnitude faster than re-evaluation. Finally, to showcase its practical benefits, we implemented a watch party use case where Incremunica efficiently maintains query results across decentralized data spaces, handling dynamic updates and new data sources seamlessly. Incremunica lays the groundwork for advancing incremental query processing for dynamic, real-time Semantic Web applications.

The future of this work, next to the future directions already mentioned in Section 3, is to evaluate the performance of different Web interfaces in the scope of IVM. Furthermore, currently, Incremunica uses a predefined incremental operator for its incremental query execution. We want to be able to choose the most performant operator for that situation automatically. Finally, Incremunica provides a test bed for research on incremental LTBQP, where the query results for a LTBQP query are maintained under changes in the data. Incremunica like Comunica, will receive continued maintenance (`https://github.com/comunica/comunica/wiki/Sustainability-Plan`).

Acknowledgement. This work was partly funded by the SolidLab Vlaanderen project (Flemish Government, EWI and RRF project VV023/10) and the FWO Project FRACTION (Nr. G086822N). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N).

References

1. Activity streams 2.0, https://www.w3.org/TR/activitystreams-core/
2. Bluesky, https://bsky.social/about/
3. Changeset, https://vocab.org/changeset/
4. Linked data event streams, https://semiceu.github.io/LinkedDataEventStreams/
5. Linked data notifications, https://www.w3.org/TR/ldn/
6. RDF delta - replicating RDF datasets, https://afs.github.io/rdf-delta/
7. Solid notifications protocol, https://solidproject.org/TR/notifications-protocol
8. Solid protocol, https://solidproject.org/TR/protocol
9. Abdullah, H., Rinne, M., Törmä, S., Nuutila, E.: Efficient matching of SPARQL subscriptions using rete. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. pp. 372–377. ACM. https://doi.org/10.1145/2245276.2245348, https://dl.acm.org/doi/10.1145/2245276.2245348
10. Ahmad, Y., Kennedy, O., Koch, C., Nikolic, M.: DBToaster: Higher-order delta processing for dynamic, frequently fresh views, http://arxiv.org/abs/1207.0137
11. Angles, R., Antal, J.B., Averbuch, A., Birler, A., Boncz, P., Búr, M., Erling, O., Gubichev, A., Haprian, V., Kaufmann, M., Pey, J.L.L., Martínez, N., Marton, J., Paradies, M., Pham, M.D., Prat-Pérez, A., Püroja, D., Spasić, M., Steer, B.A., Szakállas, D., Szárnyas, G., Waudby, J., Wu, M., Zhang, Y.: The LDBC social network benchmark. https://doi.org/10.48550/arXiv.2001.02299, http://arxiv.org/abs/2001.02299
12. Bergmann, G., Horváth, A., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over EMF models. In: Model Driven Engineering Languages and Systems, vol. 6394, pp. 76–90. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-16145-2_6, http://link.springer.com/10.1007/978-3-642-16145-2_6, series Title: Lecture Notes in Computer Science
13. Bonte, P., Calbimonte, J.P., de Leng, D., Dell'Aglio, D., Della Valle, E., Eiter, T., Giannini, F., Heintz, F., Schekotihin, K., Le-Phuoc, D.: Grounding stream reasoning research https://hal.science/hal-04792478/
14. Bonte, P., Tommasini, R.: Streaming linked data: A survey on life cycle compliance. Journal of Web Semantics **77**, 100785 (2023)
15. Cheng, S., Hartig, O.: OPT+: A monotonic alternative to OPTIONAL in SPARQL **18**(1), 169–206, https://ieeexplore.ieee.org/abstract/document/10247308/, publisher: River Publishers
16. Chirkova, R., Yang, J.: Materialized views **4**(4), 295–405, https://www.nowpublishers.com/article/Details/DBS-020, publisher: Now Publishers, Inc.
17. Colby, L.S., Griffin, T., Libkin, L., Mumick, I.S., Trickey, H.: Algorithms for deferred view maintenance. In: Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96. pp. 469–480. ACM Press. https://doi.org/10.1145/233269.233364, http://portal.acm.org/citation.cfm?doid=233269.233364
18. De Brouwer, M., Bonte, P., Arndt, D., Vander Sande, M., Dimou, A., Verborgh, R., De Turck, F., Ongenae, F.: Optimized continuous homecare provisioning through distributed data-driven semantic services and cross-organizational workflows **15**(1), 9. https://doi.org/10.1186/s13326-024-00303-4, https://jbiomedsem.biomedcentral.com/articles/10.1186/s13326-024-00303-4
19. Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.D., Boncz, P.: The LDBC social network benchmark: Interactive workload. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 619–630.

ACM. https://doi.org/10.1145/2723372.2742786, https://dl.acm.org/doi/10.1145/2723372.2742786

20. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem **19**(1), 17–37. https://doi.org/10.1016/0004-3702(82)90020-0, https://www.sciencedirect.com/science/article/pii/0004370282900200
21. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: Proceedings of the 1993 ACM SIGMOD international conference on Management of data - SIGMOD '93. pp. 157–166. ACM Press. https://doi.org/10.1145/170035.170066, http://portal.acm.org/citation.cfm?doid=170035.170066
22. Haas, L., Kossmann, D., Wimmers, E., Yang, J.: Optimizing queries across diverse data sources. International Conference on Very Large Data Bases (VLDB) (1997)
23. Hartig, O.: SQUIN: a traversal based query execution system for the web of linked data. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. pp. 1081–1084. ACM. https://doi.org/10.1145/2463676.2465231, https://dl.acm.org/doi/10.1145/2463676.2465231
24. Hartig, O.: Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) The Semantic Web: Research and Applications, vol. 6643, pp. 154–169. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-21034-1_11, http://link.springer.com/10.1007/978-3-642-21034-1_11, series Title: Lecture Notes in Computer Science
25. Hartig, O., Özsu, M.T.: Walking without a map: Optimizing response times of traversal-based linked data queries (extended version), http://arxiv.org/abs/1607.01046
26. Honkola, J., Laine, H., Brown, R., Tyrkkö, O.: Smart-m3 information sharing platform. In: The IEEE symposium on Computers and Communications. pp. 1041–1046. IEEE, https://doi.org/10.1109/ISCC.2010.5546642
27. Idris, M., Ugarte, M., Vansummeren, S., Voigt, H., Lehner, W.: General dynamic yannakakis: conjunctive queries with theta joins under updates **29**(2), 619–653. https://doi.org/10.1007/s00778-019-00590-9, https://doi.org/10.1007/s00778-019-00590-9
28. Koch, C.: Incremental query evaluation in a ring of databases. In: Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. pp. 87–98. PODS '10, Association for Computing Machinery. https://doi.org/10.1145/1807085.1807100, https://doi.org/10.1145/1807085.1807100
29. Koch, C., Lupei, D., Tannen, V.: Incremental view maintenance for collection programming. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 75–90. ACM. https://doi.org/10.1145/2902251.2902286, https://dl.acm.org/doi/10.1145/2902251.2902286
30. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ráth, I., Varró, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering. pp. 1–10. ACM. https://doi.org/10.1145/2487766.2487768, https://dl.acm.org/doi/10.1145/2487766.2487768
31. Ladwig, G., Tran, T.: Linked data query processing strategies. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) The Semantic Web – ISWC 2010, vol. 6496, pp. 453–469. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-17746-0_29, https://link.springer.com/10.1007/978-3-642-17746-0_29, series Title: Lecture Notes in Computer Science

32. Ladwig, G., Tran, T.: SIHJoin: Querying remote and local linked data. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) The Semantic Web: Research and Applications, vol. 6643, pp. 139–153. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-21034-1_10, http://link.springer.com/10.1007/978-3-642-21034-1_10, series Title: Lecture Notes in Computer Science
33. Le-Phuoc, D.: Operator-aware approach for boosting performance in RDF stream processing **42**, 38–54, https://www.sciencedirect.com/science/article/pii/S1570826816300014?casa_token=Lp5io7CXfo0AAAAA:NQ54t2EDNZSM5eZ8lPwqUN9EjflfELgxistE9CoNtif1LS4c5ozG10SfSpUKHqMtr5b04SPDCQ, publisher: Elsevier
34. McSherry, F., Murray, D., Isaacs, R., Isard, M.: Differential dataflow. https://www.microsoft.com/en-us/research/publication/differential-dataflow/
35. Miranker, D.P., Depena, R.K., Jung, H., Sequeda, J.F., Reyna, C.: Diamond: A SPARQL query engine, for linked data based on the rete match p. 7, publisher: Citeseer
36. Morandi, F., Roffia, L., D'Elia, A., Vergari, F., Cinotti, T.S.: RedSib: a smart-m3 semantic information broker implementation. In: 2012 12th Conference of Open Innovations Association (FRUCT). pp. 1–13. IEEE, https://ieeexplore.ieee.org/abstract/document/8122091/
37. Ongenae, F., Bonte, P., Nelis, J., Vanhove, T., De Turck, F.: User-friendly and scalable platform for the design of intelligent IoT services: a smart office use case. In: Proceedings of the ISWC 2016 Posters & Demonstrations Track co-located with 15th International Semantic Web Conference (ISWC 2016). vol. 1690, pp. 1–4. CEUR Workshop Proceedings, https://lirias.kuleuven.be/retrieve/746169
38. Pu, X., Wang, J., Song, Z., Luo, P., Wang, M.: Efficient incremental update and querying in AWETO RDF storage system **89**, 55–75, https://www.sciencedirect.com/science/article/pii/S0169023X13001304, publisher: Elsevier
39. Rinne, M., Nuutila, E., Törmä, S.: INSTANS: High-performance event processing with standard RDF and SPARQL. In: 11th International Semantic Web Conference ISWC. vol. 914, pp. 101–104. Citeseer
40. Roffia, L., Morandi, F., Kiljander, J., D'Elia, A., Vergari, F., Viola, F., Bononi, L., Salmon Cinotti, T.: A semantic publish-subscribe architecture for the internet of things **3**(6), 1274–1296. https://doi.org/10.1109/JIOT.2016.2587380, http://ieeexplore.ieee.org/document/7505922/
41. Schmedding, F.: Incremental SPARQL evaluation for query answering on linked data. In: COLD. https://ceur-ws.org/Vol-782/Schmedding_COLD2011.pdf
42. Van de Sompel, H., Nelson, M.L., Sanderson, R., Balakireva, L.L., Ainsworth, S., Shankar, H.: Memento: Time travel for the web, http://arxiv.org/abs/0911.1112
43. Steenwinckel, B., Soete, C., Moens, P., Mussche, J., Hoecke, S.V., Ongenae, F.: Quality in color: Using knowledge graphs for enhanced quality control in an automotive paintshop. In: Demartini, G., Hose, K., Acosta, M., Palmonari, M., Cheng, G., Skaf-Molli, H., Ferranti, N., Hernández, D., Hogan, A. (eds.) The Semantic Web – ISWC 2024. pp. 236–252. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-77847-6_13
44. Szárnyas, G., Izsó, B., Ráth, I., Harmath, D., Bergmann, G., Varró, D.: IncQuery-d: A distributed incremental model query framework in the cloud. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) Model-Driven Engineering Languages and Systems, vol. 8767, pp. 653–669. Springer International Publishing. https://doi.org/10.1007/978-3-319-11653-2_40, http://link.springer.com/10.1007/978-3-319-11653-2_40, series Title: Lecture Notes in Computer Science
45. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: A modular SPARQL query engine for the web. In: Vrandečić, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti, V., Celino, I., Sabou, M., Kaffee, L.A., Simperl, E. (eds.) The

Semantic Web – ISWC 2018, vol. 11137, pp. 239–255. Springer International Publishing. https://doi.org/10.1007/978-3-030-00668-6_15, https://link.springer.com/10.1007/978-3-030-00668-6_15, series Title: Lecture Notes in Computer Science

46. Taelman, R., Vander Sande, M., Verborgh, R., Mannens, E.: Versioned triple pattern fragments: a low-cost linked data interface feature. In: 3rd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW 2017) and the 4th Workshop on Linked Data Quality (LDQ 2017) co-located with 14th European Semantic Web Conference (ESWC 2017). pp. 1–11. https://biblio.ugent.be/publication/8540854/file/8540856.pdf
47. Taelman, R., Verborgh, R.: Link traversal query processing over decentralized environments with structural assumptions. In: Payne, T.R., Presutti, V., Qi, G., Poveda-Villalón, M., Stoilos, G., Hollink, L., Kaoudi, Z., Cheng, G., Li, J. (eds.) The Semantic Web – ISWC 2023, vol. 14265, pp. 3–22. Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-47240-4_1, https://link.springer.com/10.1007/978-3-031-47240-4_1, series Title: Lecture Notes in Computer Science
48. Tailhardat, L., Chabot, Y., Troncy, R.: Designing NORIA: a knowledge graph-based platform for anomaly detection and incident management in ICT systems. In: KGCW@ ESWC. https://ceur-ws.org/Vol-3471/paper3.pdf
49. Tommasini, R., Bonte, P., Ongenae, F., Della Valle, E.: Rsp4j: an api for rdf stream processing. In: The Semantic Web: 18th International Conference, ESWC 2021, Virtual Event, June 6–10, 2021, Proceedings 18. pp. 565–581. Springer (2021)
50. Van De Vyvere, B., Colpaert, P., Verborgh, R.: Comparing a polling and push-based approach for live open data interfaces. In: Bielikova, M., Mikkonen, T., Pautasso, C. (eds.) Web Engineering, vol. 12128, pp. 87–101. Springer International Publishing. https://doi.org/10.1007/978-3-030-50578-3_7, http://link.springer.com/10.1007/978-3-030-50578-3_7, series Title: Lecture Notes in Computer Science
51. Verborgh, R., Taelman, R.: Guided link-traversal-based query processing, http://arxiv.org/abs/2005.02239
52. Verborgh, R., Vander Sande, M., Hartig, O., Van Herwegen, J., De Vocht, L., De Meester, B., Haesendonck, G., Colpaert, P.: Triple pattern fragments: a low-cost knowledge graph interface for the web **37**, 184–206, https://doi.org/10.1016/j.websem.2016.03.003, publisher: Elsevier
53. Verstichel, S., Blommaert, T., Coppens, S., Van Maele, T., Haerick, W., Ongenae, F.: Harmoney: Semantics for FinTech:. In: Proceedings of the 12th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management. pp. 141–148. SCITEPRESS - Science and Technology Publications. https://doi.org/10.5220/0010061301410148, https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0010061301410148